CHAPTER 1

Introducing Ajax

The Internet as we know it today has undergone tremendous change. Beginning with simple textual browsers that allowed scientists to exchange research, the Internet is now a hub for commerce and information. Over that time, we've seen a number of new technologies and approaches—from the earliest graphical browsers to podcasts. Today, the Internet has become the leading platform for numerous applications (when was the last time you actually spoke with a travel agent?), but despite the convenience, few would mistake a Web application with its desktop cousin. This chapter will give a brief overview of the evolution of Web applications. Once we have you grounded in the past, we'll introduce you to what we view as the future: Ajax.

A Short History of Web Applications

In the beginning, it was all so simple. Initially connecting a handful of top research institutions in the United States, the original "Internet" was designed to facilitate the sharing of scientific research. Whether you were a librarian, nuclear physicist, or computer scientist, you had quite a complex system to learn—Firefox and Internet Explorer weren't even concepts when J.C.R. Licklider of the Massachusetts Institute of Technology (MIT) first presented his ideas on a "Galactic Network" in 1962.

Licklider went on to head up computer research at the Defense Advanced Research Projects Agency (DARPA), where he preached the importance of his networking ideas. About the same time, Leonard Kleinrock and Lawrence G. Roberts of MIT were working on packet-switching theory, a key concept to networking computers. Roberts went on to create, with Thomas Merrill, the first wide area network in 1965 when he connected a TX-2 in Massachusetts with a Q-32 in California over a dial-up connection.

Roberts took the results of his experiments to DARPA in late 1966 where he designed his plan for the Advanced Research Projects Administration Network (ARPANET). By now, Kleinrock was at the University of California–Los Angeles' Network Measurement Center, which was selected as the first node of ARPANET and where in 1969 Bolt Beranek and Newman (BBN) installed the first packet switches called Interface Message Processors (IMPs). The Stanford Research Center was selected as the second node, and in October 1969 the first host-to-host messages were exchanged. Shortly thereafter, the University of California–Santa Barbara and the University of Utah were added as nodes, beginning what we know today as the Internet.

Minicomputers were just starting to appear: Digital Equipment Corporation created the PDP-1, which was followed by the tremendous success of the PDP-8, the PDP-11, and the VAX-11/780. Computing power was becoming increasingly affordable—no longer were we begging time on a handful of massive mainframe computers. Computing was becoming more democratic; still, we had yet to see the personal computer revolution.

Originally researchers thought Transmission Control Protocol (TCP) would work only with the large systems for which it was designed. However, David Clark's research team at MIT proved that workstations could be networked along with big iron. Clark's research, combined with the personal computing explosion of the 1980s and 1990s, paved the way for the always-on world in which we currently reside.

During the 1980s, a number of changes occurred. As the number of hosts grew from just a few to thousands, they were assigned names so people didn't have to memorize the numeric addresses. This switch, combined with the increasing number of hosts, gave birth to the Domain Name System (DNS). Further, ARPANET was transitioning from using Network Control Protocol (NCP) to Transmission Control Protocol/Internet Protocol (TCP/IP), the standard used by the military. By the mid-1980s, the Internet was established as a platform to connect disparate groups of researchers, and other networks began to appear: National Aeronautics and Space Administration created SPAN, the U.S. Department of Energy established MFENet for research on Magnetic Fusion Energy, and a grant from the National Science Foundation helped create the CSNET for computer science research.

In 1989, Tim Berners-Lee of the European Council for Nuclear Research (CERN) came up with an interesting concept. He thought, rather than merely reference another work, why not actually link to it? While reading one paper, a scientist could simply open a referenced paper. The term *hypertext* was in vogue, and drawing upon his previous work in document and text processing, Berners-Lee invented a subset of Standard Generalized Markup Language (SGML) called HyperText Markup Language (HTML). The beauty of HTML is that it separates the information about how text should be rendered from the actual implementation of the display. Along with creating the simple protocol called HyperText Transfer Protocol (HTTP), Berners-Lee invented the first Web browser, called WorldWideWeb.

Browser History

Until the recent ascent of alternatives such as Firefox, Safari, and Opera, most people associated the Web browser with Microsoft's ubiquitous Internet Explorer. Despite what many newcomers may think, Internet Explorer wasn't even close to being the first browser on the market. In fact, Berners-Lee created the first Web browser (originally called WorldWideWeb but later renamed to Nexus) on and for the NeXT computer, and he released it to personnel at CERN in 1990. Berners-Lee and Jean-Francois Groff ported WorldWideWeb to C, renaming the browser to libwww. The early 1990s saw a number of browsers, including the line-mode browser written by Nicola Pellow (which allowed users of any system from Unix to Microsoft DOS to access the Internet) and Samba, the first browser for the Macintosh.

In February 1993, Marc Andreessen and Eric Bina of the National Center for Supercomputing Applications at the University of Illinois–Urbana-Champaign released Mosaic for Unix. A few months later, Mosaic became the first cross-platform browser when Aleks Totic released a version for the Macintosh. It quickly spread and became the most popular Web browser.[1] The technology was licensed to Spyglass, where it was further licensed to Microsoft for use in Internet Explorer.

---

1. One author recalls his first introduction to Mosaic: "I had just been introduced to Lynx and was, as a freshman chemistry student, amazed I could browse the stacks of Oxford from central Minnesota (albeit via text-based browsing only). After seeing a beta version of Mosaic and noticing how slow and choppy the experience was, I vowed to stick with Lynx, and I'm proud to say I use Firefox today."

Developers at the University of Kansas wrote a text-based browser called Lynx in 1993 that became the standard for character terminals. A team in Oslo, Norway, in 1994 developed Opera, which was made widely available in 1996. In December 1994, Netscape released the 1.0 version of Mozilla, and the first for-profit browser was born. In 2002, an open-source version was released that grew into the popular Firefox browser, released in November 2004.

When Microsoft released Windows 95, it included Internet Explorer 1.0 as part of its Microsoft Plus! pack. Despite its integration with the operating system, most people stuck with Netscape, Lynx, or Opera. Version 2.0 made significant strides by adding support for cookies, Secure Socket Layer (SSL), and other emerging standards. The second version was also available for the Macintosh, making it the first cross-platform browser from Microsoft. Still, most users stuck with what they were using.

However, in the summer of 1996, Microsoft released version 3.0. Virtually overnight, people flocked to Internet Explorer. Of course, it didn't hurt that Netscape charged money for its browser and Microsoft offered Internet Explorer for free. The Internet community was polarized on the issue of browser dominance, as many feared Microsoft would do to the Web what it did to the desktop. Some were concerned about security, and sure enough, nine days after it was released, the first security problem was reported. By 1999's release of Internet Explorer 5, it was the most widely used browser.

The Evolution of Web Applications

At first, all Web pages were static; users requested a resource, and the server returned it. Nothing moved, nothing flashed. Frankly, for a great number of Web sites, this was just fine—Web pages were nothing more than electronic copies of text that was at one point bound and distributed. In the early days of the browser, the static nature of Web pages wasn't an issue; scientists were using the Internet to exchange research papers, and universities were posting class information online. Businesses hadn't yet figured out what to do with this new "channel." In fact, at first, corporate home pages often displayed little more than contact information or some documentation. However, it didn't take long for Web users to want a more dynamic experience. The personal computer was a stalwart of business, and from dorm rooms to home offices more and more computers were starting to appear. With the advent of Windows 95 and the rich experience of thick applications such as Corel WordPerfect and Microsoft Excel, users' expectations were rising.

CGI

The first solution to making the Web more dynamic was Common Gateway Interface (CGI). Unlike static Web retrieval, CGI allows you to create programs that execute when a user makes a request. Say you want to display items for sale on your Web site—with a CGI script you can access your product database and display the results. Using simple HTML forms and CGI scripts, you can create a simple storefront that allows you to sell products to anyone with a browser. You can write CGI scripts in any number of languages from Perl to Visual Basic, making the scripts available to a wide range of skill sets.

However, CGI is not the safest approach for creating dynamic Web pages. With CGI, you literally let people execute a program on your system. Most of the time this probably isn't a problem, but a user with malicious intent can exploit this and cause your system to run something you didn't intend. Despite that drawback, CGI continues to be used today.

Applets

It was clear that CGI could be improved upon. In May 1995, John Gage of Sun and Andreessen (now of Netscape Communications Corporation) announced the birth of a programming language called Java. Netscape Navigator would offer support for this new language, originally intended for set-top boxes (and you thought Microsoft and Sony were the first companies to fight for control of your living room!). As is often the case when something truly revolutionary happens, Java and the Internet were at the right place at the right time, and within a few months of its release on the Web, thousands of people had downloaded Java. With Netscape's dominant Navigator supporting Java, a new avenue for dynamic Web pages had opened: the era of applets had begun.

Applets allow developers to write small applications that can be embedded on a Web page. As long as they use a Java-aware browser, users can run the applets in the browser's Java Virtual Machine (JVM). While applets can do a number of things, they have some restrictions: they are typically prevented from reading or writing to the file system, they cannot load native libraries, and they may not start programs on the client. In addition to these restrictions, applets run with a sandbox security model that helps protect users from malicious code.

For many people, applets were their first exposure to the Java programming language, and at the time they were an excellent way to create dynamic Web applications. Applets let you create a "thick" client inside your browser, within the security constraints of the platform. In some areas at the time, applets were widely used; however, they never really grabbed the Web community.[2] One problem was familiar to developers of thick clients: you have to deploy the proper Java version to the client. Because applets run in the virtual machine of a browser, developers have to make sure the client has the proper version of Java installed. Though not insurmountable, this issue greatly compromised the adoption of applet technology. It didn't help that poorly written applets caused complications on client machines and made many customers hesitant to use applet-based solutions. For those of you unfamiliar with applets, Figure 1-1 shows a clock applet from Sun.

2. The corporate time-tracking software of one author's employer is, believe it or not, an applet.

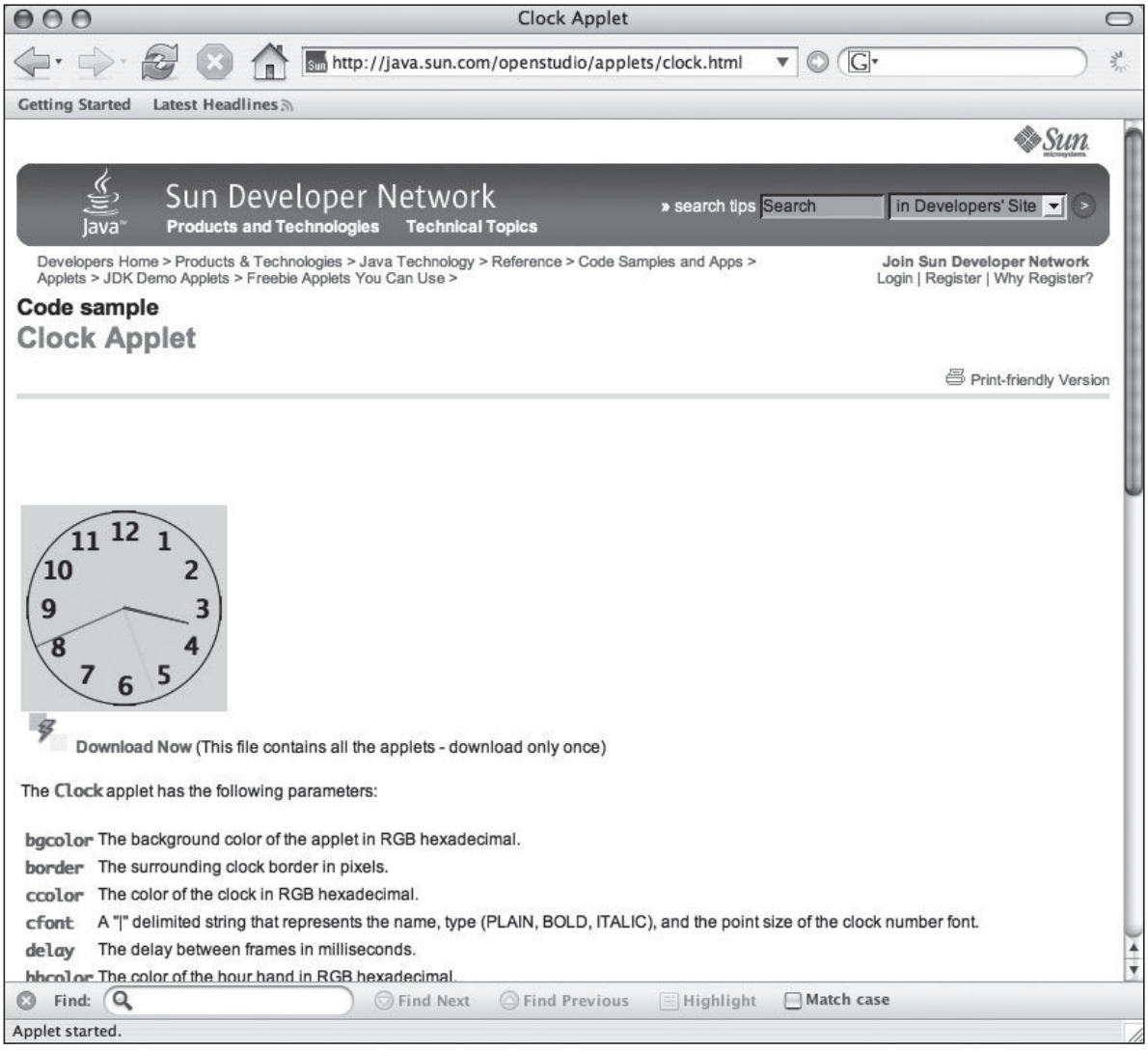

Figure 1-1. *A clock applet from Sun*

JavaScript

About this same time, Netscape created a scripting language eventually called JavaScript. (It was called Mocha when prototyped and then LiveWire and LiveScript before being released as JavaScript.) JavaScript was designed as a way of making applets easier to develop for Web designers and programmers who weren't familiar with Java. (Of course, Microsoft had its own answer to JavaScript—a scripting language called VBScript.) Netscape hired Brendan Eich to design and implement the new language, and he thought a dynamically typed scripting language was just what was needed. Though it has been much maligned for its lack of development tools, useful error messages, and debuggers, JavaScript is a powerful way to create dynamic Web applications.

Originally, JavaScript was created to help developers dynamically modify the tags on their pages in an effort to provide a richer client experience. It became evident that one could treat the page as an object, and thus was born the Document Object Model (DOM). At first, JavaScript and the DOM were tightly intertwined, but eventually they evolved into separate

constructs. The DOM is a fully object-oriented representation of the page that can be modified with a scripting language such as JavaScript or VBScript.

Eventually, the World Wide Web Consortium (W3C) got involved and standardized the DOM, while the European Computer Manufacturers Association (ECMA) ratified JavaScript as the ECMAScript specification. Any page and script written according to these standards should look and behave identically in any browser that adheres to these guidelines.

A number of factors conspired against JavaScript in its early years. Browser support was spotty (even today the same script may behave differently across browsers), and clients are free to turn JavaScript off (some well-publicized security breaches have prompted many users to do so). The difficulty of developing JavaScript (can you say *alert*?) caused many developers to shy away from using the language often, while other developers simply ignored JavaScript, considering it a toy language for graphics designers. Most were content to create simple form-based applications after experiencing extreme mental fatigue while attempting to use, test, and debug complex JavaScript.

Servlets and ASPs and PHP . . . Oh My!

Though Web-based, applets still presented many of the issues associated with thick client applications. In the era of the dial-up connection (still far too prevalent even today), downloading the entire code base for a complex applet could take more time than a user was willing to invest. Developers also had to worry about the version of Java present on the client, and some virtual machines left something to be desired.[3] Ideally, you just serve up static Web pages—after all, that was what the Internet was truly designed to do. Of course, static pages are, well, static, but if you could *dynamically* generate content on the server and *return* static content, that would get you somewhere.

Within a year of Java being introduced to the world, Sun introduced servlets. No longer would your Java code run in the client browser as with applets; it would run on an application server that you controlled. This would allow developers to leverage existing business applications, and if you needed to upgrade to the latest Java version, you had to worry only about your server. Java's "write once, run anywhere" nature allowed developers to select best-of-breed application servers and server environments—yet another advantage of the new technology. Servlets also served as an alternative to CGI scripts.

Servlets were a huge step forward. They offered full access to the entire set of Java application programming interfaces (APIs) and came with a complete library for handling HTTP. However, servlets weren't perfect. Interface design with servlets can be hard. In a typical servlet interaction, you get some information from your user, perform some business logic, and then, using what amounts to print lines, create the HTML to display for the user. Code like that shown in Listing 1-1 was common.

3. The Microsoft virtual machine never supported Java after the 1.1 specification, greatly hindering what applets could do on the Microsoft platform.

Listing 1-1. *Simple Servlet Code*

```
response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet SimpleServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World</h1>");
        out.println("<p>Imagine if this were more complex.</p>");
        out.println("</body>");
        out.println("</html>");

        out.close();
```

This small amount of code produces the rather simple Web page shown in Figure 1-2.

Figure 1-2. *Output from the simple servlet in Listing 1-1*

Besides being error prone and difficult to visualize, servlets had a skill set mismatch. In general, the people writing the server-side code were software developers schooled in algorithms and compilers, not the graphics designers who were crafting elegant corporate Web sites. Business developers were focused not only on writing business logic but also had to worry about creating coherent designs. A separation of presentation and business logic was needed. What we needed were JavaServer Pages (JSPs).

JSPs were, to an extent, a response to Microsoft's Active Server Pages (ASPs). Microsoft learned from the mistakes Sun made with the Servlet specification and created ASPs to simplify creating dynamic pages. Microsoft added excellent tool support and tight integration with its Web server. JSPs and ASPs are similar in that they were designed to separate the business processing from the presentation layout. Some technical differences exist (Sun too learned from Microsoft), but both allow Web designers to focus on the layout while software developers focus on the business logic. Listing 1-2 shows a simple JSP.

Listing 1-2. *Simple JSP*

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Hello World</title>
    </head>
    <body>

    <h1>Hello World</h1>
    <p>This code is more familiar for Web developers.</p>

    </body>
</html>
```

This code produces the output shown in Figure 1-3.

Figure 1-3. *Output from a simple JSP*

Of course, Microsoft and Sun don't own a monopoly on server-side solutions. A number of other options exist, from PHP to ColdFusion. Amazing tools drive some developers; others are looking for simpler languages. At the end of the day, all these solutions perform the same task—they dynamically generate HTML. Generating content on the server solves the distribution problem; however, the user experience that is possible with raw HTML pales in comparison with what you can do with a thick client or applets. The following sections cover a number of other solutions that were created in an effort to provide a richer user experience.

Flash

Microsoft and Sun weren't the only companies looking to solve the dynamic Web page problem. In the summer of 1996 FutureWave released a product called FutureSplash Animator. Growing out of a Java-based animation player, FutureWave soon sold its company to Macromedia, which rebranded the product as Flash.

Flash allows designers to create amazing applications that are highly dynamic in nature. Companies can deliver highly interactive applications on the Web that are almost indistinguishable from their thick client brethren (see Figure 1-4). Unlike applets, servlets, and CGI scripts, Flash does not require programming skills and is easy to learn. In the go-go days of the late 1990s, this was a significant plus, as many employers scrambled to find employees with the requisite skills. However, this ease of use comes at a cost.

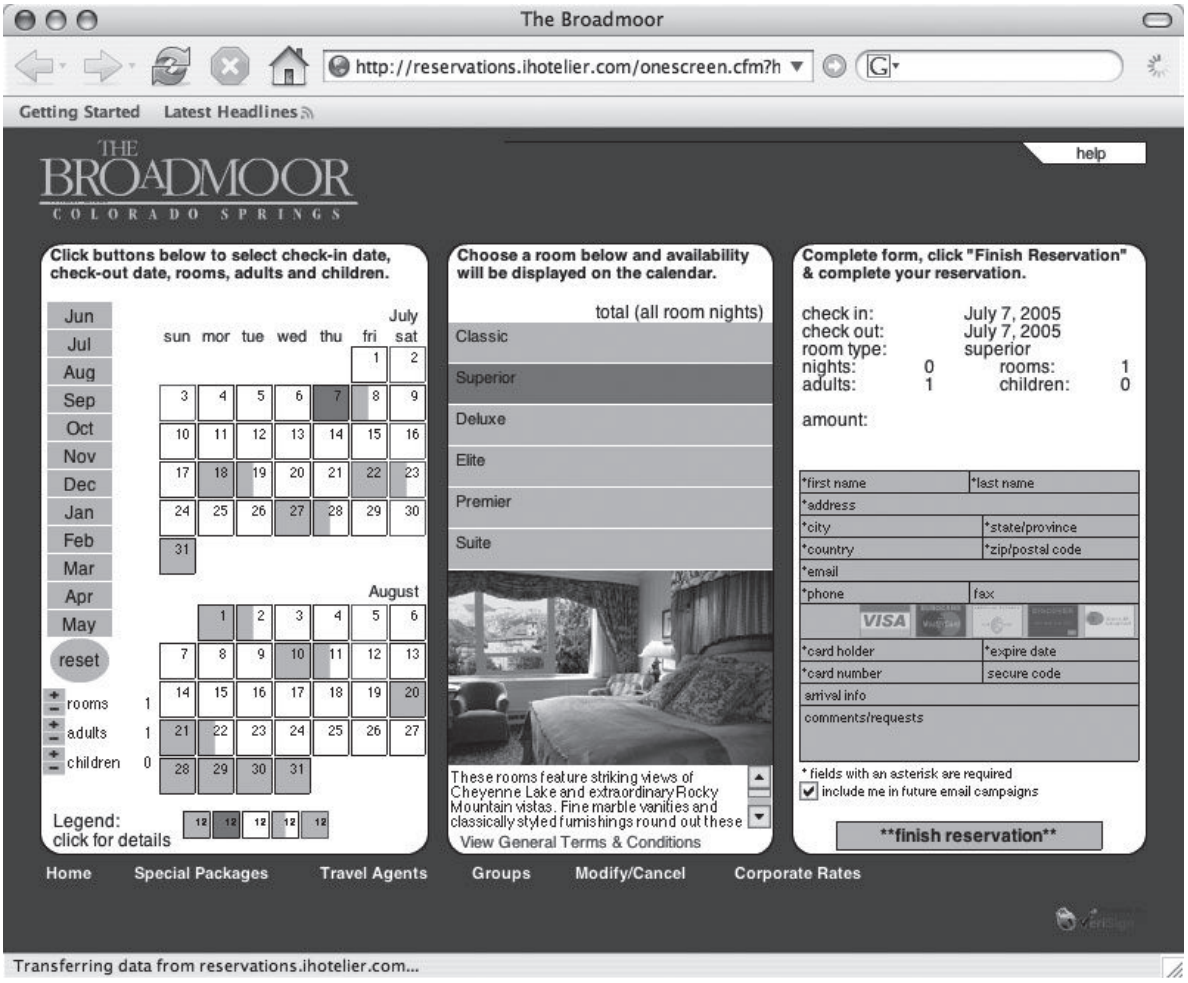

Figure 1-4. *A Flash application*

Like many solutions, Flash requires software on the client. Though the required Shockwave Player plug-in shipped with several popular operating systems and browsers, it was not universal. Despite the free download, fear of viruses caused many users to refuse the install of the software. Flash applications can also require significant amounts of network bandwidth to perform properly, and the lack of widespread broadband connections limited the adoption of

Flash (and thus was born the "skip intro" link). While some sites choose to have multiple versions of their Web application tailored to various connection speeds, many companies could not justify the added development costs of supporting two or three catalog sites.

In sum, creating Flash applications requires proprietary software and browser plug-ins. Unlike applets that can be written with text editors and a free Java Development Kit (JDK), using the complete Flash toolkit costs hundreds of dollars per seat. Though not insurmountable, these factors have slowed the widespread adoption of Flash for dynamic Web applications.

The DHTML Revolution

When Microsoft and Netscape released version 4 of their respective browsers, Web developers had a new option: dynamic HTML (DHTML). Despite what many think, DHTML is not a W3C standard; it's really more of a marketing tool. In reality, it's a combination of HTML, Cascading Style Sheets (CSS), JavaScript, and the DOM. The combination of these technologies allows developers to modify the content and structure of a Web page on the fly.

The initial buzz was very positive for DHTML. However, it required browser versions that weren't yet widely adopted. Though both Internet Explorer and Netscape supported DHTML, their implementations were different, meaning developers had to know which browser their clients were using. Often this meant having lots of code that checked for browser type and version, which further increased the cost of development. Some were hesitant to try this approach because of the lack of an official standard for DHTML. Still, the possibilities were there for something new to come along.

The XML Derivatives

Since its invention in the mid-1990s, the W3C's eXtensible Markup Language (XML) derivative of SGML has been amazingly popular. Seen by many as the answer to all that ails computer development, XML shows up everywhere. In fact, Microsoft has announced that Office 12 will support XML file formats.

Today we have no fewer than four XML derivatives for creating Web applications (and that doesn't count the W3C's XHTML): XUL from Mozilla; XAMJ, an open-source alternative that blends Java into the mix; MXML from Macromedia; and XAML from Microsoft.

XUL: XUL (pronounced "zool") stands for XML User Interface Language and comes from the Mozilla Foundation. The popular Firefox browser and Thunderbird mail client are written in XUL. XUL lets developers build rich applications that can run with or without a connection to the Internet. Designed to be learned quickly by developers familiar with DHTML, XUL provides cross-platform support for standard interface widgets such as windows and buttons. Though not a standard itself, XUL is based on standards such as HTML 4.0, CSS, the DOM, XML, and ECMAScript. XUL applications can be run from a browser or installed on a client machine.

Of course, XUL is not without its drawbacks. XUL requires the Gecko engine, and as of now Internet Explorer has no corresponding plug-in. Though Firefox has certainly captured a respectable share in the browser usage statistics, the lack of Internet Explorer support largely makes XUL unusable for most applications. Several projects are underway to make XUL available to a number of platforms, including Eclipse.

XAML: XAML (pronounced "zammel") is a component of Microsoft's upcoming operating system code-named Vista. XAML is short for eXtensible Application Markup Language and defines the standard for creating user interfaces in Vista. Similar to HTML, XAML uses tags to create standard elements such as buttons and text boxes. Based on top of Microsoft's .NET platform, XAML is compiled into .NET classes.

It should be pretty clear what the limitations of XAML are. As a Microsoft product, you are relegated to a Microsoft operating system. In many cases (especially corporations), this may not be problematic, but no bricks-and-mortar company could justify turning away paying customers simply because they didn't drive a particular model of automobile, for example. Combined with the continually shifting ship date of Vista, XAML isn't much of a player right now. That said, in a few years, we might be whistling a different tune.

MXML: Macromedia created MXML as the markup language for use with its Flex technology; MXML stands for Maximum eXperience Markup Language. MXML is designed to be like HTML, allowing you to design your interface in a declarative manner. Like XUL and XAML, MXML provides a richer set of interface components, such as DataGrid and TabNavigator, that allow you to create rich Internet applications. MXML doesn't stand on its own, though; it relies on Flex and the ActionScript programming language to code business logic.

MXML has some of the same limitations that Flash does. It's proprietary and relies on expensive development and deployment environments. Though .NET support is expected in the future, today Flex runs only on top of Java 2 Enterprise Edition (J2EE) application servers such as Tomcat and IBM's WebSphere, further limiting its adoption.

XAMJ: Not to be outdone, the open-source community has added an entry to the XML derivative world of interface design. XAMJ was recently introduced as another cross-platform option in the Web application developer's toolkit. This particular derivative is based on Java, which provides the full power of one of the most popular object-oriented languages in use today. XAMJ is essentially an alternative to XAML- or HTML-based applications that seeks to be a more secure option that neither is reliant on a particular framework nor requires a high-speed Internet connection. XAMJ is a compiled language that builds upon a "clientlet" architecture, and though stand-alone applications are possible, in general XAMJ-based programs will be Web based. As of this writing, XAMJ is too new to properly critique; however, it bears watching.

As long as we are talking about "things that start with *X*," let's not forget the W3C XForms specification. XForms is designed to support a richer user interface while decoupling data from presentation. Not surprisingly, XForms data is XML, which allows you to use existing XML technologies such as XPath and XML Schema. XForms can do anything standard HTML can do plus more, including checking field values on the fly and integrating with Web Services. Unlike many W3C specifications, XForms doesn't require a new browser—you can use several existing implementations now. Like most of the XML derivatives, XForms is a fresh approach, so patience may be appropriate.

The Fundamental Problem

Where does that leave you? For all but the most demanding of client applications, the Web has become the platform of choice. While it's obvious that Web-based applications are easy to deploy, the low barrier of entry for users might be their greatest strength. With the ubiquity of the browser and no need to download and install new software, browser-based clients make it simple for users to try a new application. Users are far more likely to simply click a link and try your application than go through the effort of a multiple-megabyte installation. Browser-based applications are also operating system agnostic, which means not only can you reach those people running Linux and OS X, but you don't have to worry about developing and maintaining multiple installation packages.

If Web-based applications are the greatest thing since sliced bread, why did we write this book? If you look to the origins of the Internet, you see a world of scientists and academics exchanging papers and research—a simple request/response paradigm. It had no need for conversational state, and it had no need for a shopping cart; people were simply exchanging documents. While you have long had a number of options for creating dynamic Web-based applications, if you want to truly reach the largest body of users, you have to stay pretty close to the browser, which means you are always held back by the synchronous nature of the request/response underpinnings of the Internet.

Compared to thick client applications such as Microsoft Word or Intuit's Quicken, the Web model was certainly an adjustment for the average user. However, the ease of deployment and the ascent of the browser meant most users learned to adapt. Still, many thought the Web offered applications that were second-class citizens with inferior user experiences. Because the Internet is a synchronous request/response system, the entire page was constantly refreshing in the browser. Originally, it didn't matter how simple the request was—if the user made one or two changes, the entire document had to be sent back to the server and the entire page was repainted. Though usable, this total refresh limitation meant applications were rather crude.

That didn't mean developers simply sat on their hands and accepted the status quo. Microsoft knew a thing or two about interactive applications. Microsoft was never satisfied with the limitations of the standard paradigm, and therefore it introduced the concept of remote scripting. The magic of remote scripting is simple: it allows developers to create pages that interact with the server in an asynchronous manner. For instance, a customer could select their state from a drop-down list, causing a script to run on the server and determine their shipping costs. More important, these costs could then be displayed *without the entire page refreshing*! Of course, Microsoft's solution works with its technology only and requires Java, but this advancement showed that richer browser applications were possible.

Other solutions exist to the synchronous page refresh problem. Brent Ashley developed a platform-neutral answer to Microsoft's remote scripting when he created JavaScript Remote Scripting (JSRS). JSRS relies on a client-side JavaScript library and DHTML to make asynchronous calls to the server. At the same time, many people took advantage of the IFRAME tag to load only portions of the page or make "hidden" calls to the server. Though a workable solution used by many, it certainly wasn't ideal—it was really a bit of a hack.

Ajax

So here we are: clients want a more full-featured application, and developers want to avoid deploying executables to thousands of workstations. We've tried a number of alternatives, but none has been the panacea it was touted as. However, recent developments have added an incredibly powerful tool to our design kit.

Today we have another option, another tool, to create truly rich browser-based applications. Today we have Ajax. Ajax is more of a technique than it is a specific technology, though the aforementioned JavaScript is a primary component. We know you're saying, "JavaScript is not worth it," but with the resurgent interest in the language because of Ajax, application and testing frameworks, combined with better tool support, are easing the burden on developers. With the introduction of Atlas, Microsoft is throwing its weight firmly behind Ajax, while the infamous Rails Web framework comes prebuilt with outstanding Ajax support. In the Java space, Sun has added several Ajax components to its BluePrints Solutions Catalog.

Honestly, Ajax isn't anything new. In fact, the "newest" technology related to the term—the XMLHttpRequest object (XHR)—has been around since Internet Explorer 5 (released in the spring of 1999) as an Active X control. What is new, however, is the level of browser support. Originally, the XHR object was supported in only Internet Explorer (thus limiting its use), but starting with Mozilla 1.0 and Safari 1.2 support is widespread. The little-used object and the basic concepts are even covered in a W3C standard: the DOM Level 3 Load and Save Specification. At this point, especially as applications such as Google Maps, Google Suggest, Gmail, Flickr, Netflix, and A9 proliferate, XHR is becoming a de facto standard.

Unlike many of the approaches mentioned in the previous pages, Ajax works in most modern browsers and doesn't require any proprietary software or hardware. In fact, one of the real strengths of this approach is that developers don't need to learn some new language or scrap their existing investment in server-side technology. Ajax is a client-side approach and can interact with J2EE, .NET, PHP, Ruby, and CGI scripts—it really is server agnostic. Short of a few minor security restrictions, you can start using Ajax right now, leveraging what you already know.

"Who is using Ajax?" you may ask. As mentioned, Google is clearly one of the early adopters, with several examples of the technology, including Google Maps, Google Suggest, and Gmail, to name just a few applications. Yahoo! is beginning to introduce Ajax controls, and Amazon has a neat search tool that uses the technique extensively—moving the sliders for a given facet of a diamond results in dynamically updated results (see Figure 1-5). The page isn't refreshed each time you change your criteria, and the server is queried while you move a slider, allowing you to narrow your options both quickly and easily.

Netflix, the popular DVD rental company, uses Ajax to provide a greater depth of information as users browse for movies. When a customer hovers over the graphic for a movie, the movie ID is sent to their central servers, and a bubble appears that provides more details about the movie (see Figure 1-6). Again, the page is not refreshed, and the specifics for each movie aren't found in hidden form fields. This approach allows Netflix to provide more information about its movies without cluttering its pages. It also makes browsing easier for their customers—they don't have to click the movie and then click back to the list; they simply have to hover over a movie! We want to stress that Ajax isn't limited to "dot-com" darlings; corporate developers are starting to scratch the surface as well, with many using Ajax to solve particularly ugly validation situations or to retrieve data on the fly.

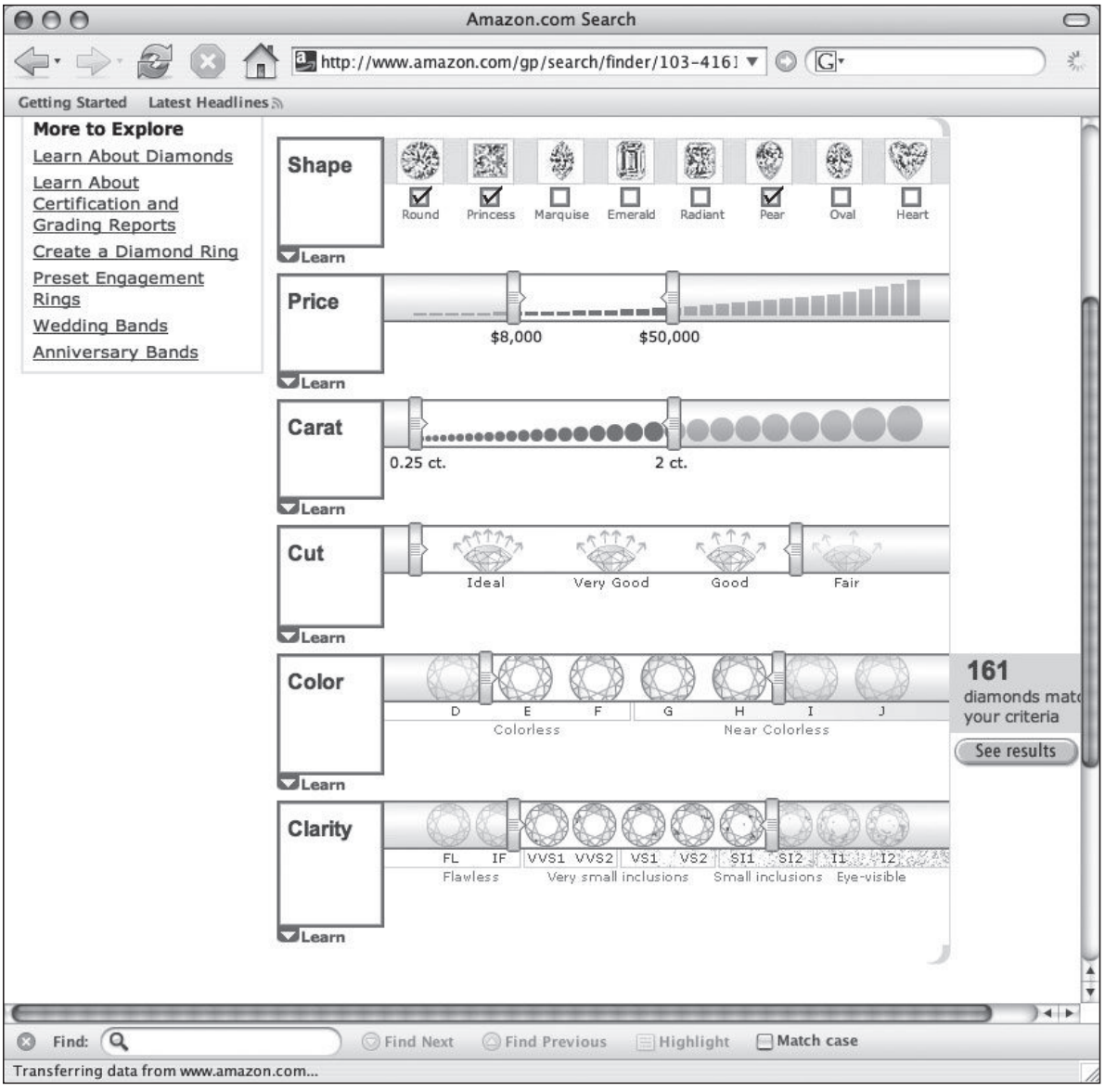

Figure 1-5. *Amazon's diamond search*

If it isn't exactly new, the *approach* that is the meat of Ajax is an important shift in the Internet's default request/response paradigm. Web application developers are now free to interact with the server asynchronously, meaning they can perform many tasks that before were limited to thick clients. For example, when a user enters a zip code, you can validate it and populate other parts of a form with the city and state; or when they select United States, you can populate a state drop-down list. We've been able to mimic these approaches before, but it's much simpler to do with Ajax.

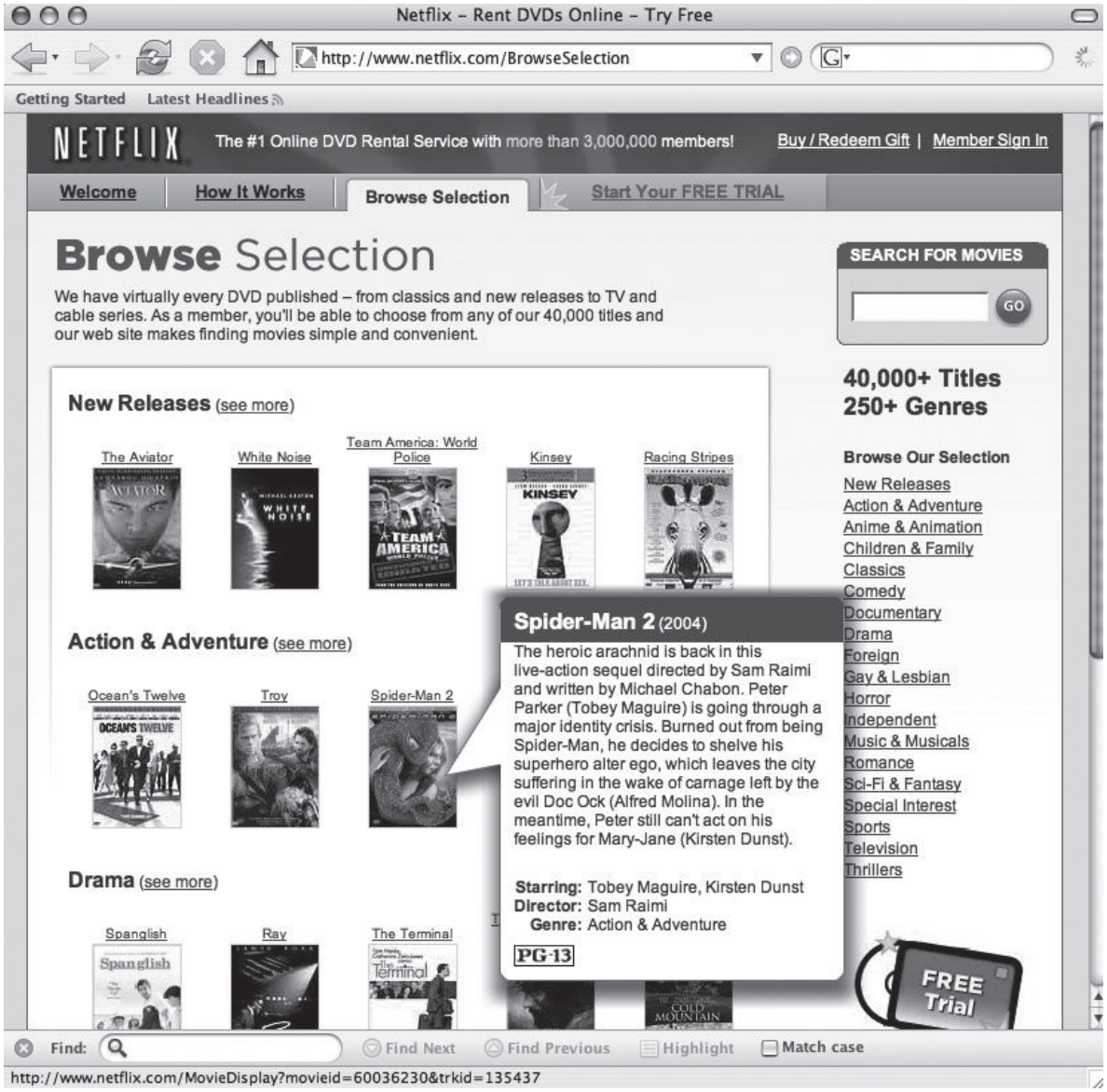

Figure 1-6. *Netflix's browse feature*

So, who invented Ajax? The exact origin involved is a subject of debate; however, Jesse James Garrett of Adaptive Path first coined the term in February 2005. In his essay "Ajax: A New Approach to Web Applications," Garrett discusses how the gap is closing between thick client, or desktop, applications and thin client, or Web, applications. Of course, Google really gave the techniques a high profile when it released Google Maps and Google Suggest in Google Labs; also, there have been numerous articles on the subject. But Garrett gave us a term that wasn't quite as, shall we say, wordy as Asynchronous, XMLHttpRequest, JavaScript, CSS, the DOM, and so on. Though originally considered an acronym for Asynchronous JavaScript + XML, the term is now used simply to encompass all the technologies that allow a browser to communicate with the server without refreshing the current page.

We can hear you saying, "So, what's the big deal?" Well, using XHR and working asynchronously with the server lets you create Web applications that are far more dynamic. For example, say you have a drop-down that is filled based on the input in some other field or drop-down. Ordinarily, you would have to send all the data down to the client when the page first loaded and use JavaScript to populate your drop-down based on the input. It's not hard to do, but it does bloat the size of your page, and depending on just how dynamic that drop-down list is, size could be an issue. With Ajax, when the trigger field changes or the focus is lost, you can make a simple request to the server for only the information you need to update your drop-down.

Imagine the possibilities for validation alone. How many times have you written some JavaScript validation logic? While the edit might be simple in Java or C#, the lack of decent debuggers, combined with JavaScript's weak typing, can make writing them in JavaScript a real pain and error prone. How often do these client-side validation rules duplicate edits on the server? Using XHR, you can make a call to the server and fire *one* set of validation rules. These rules can be far richer and more complex than anything you would write in JavaScript, and you have the full power of debuggers and integrated development environments (IDEs).

We can hear some of you now: "I've been doing that for years with IFRAMES or hidden frames." We've even used this particular technique as a way to post or refresh parts of a page instead of the entire browser, and truth be told, it works. However, many would consider this approach a hack to get around XHR's original lack of cross-browser support. The XHR object that is the heart of Ajax is truly designed to allow asynchronous retrieval of arbitrary data from the server.

As we've discussed, traditional Web applications follow a request/response paradigm. Without Ajax, the entire page (or with IFRAMEs, parts of the page) is reloaded with each request. The previously viewed page is reflected in the browser's history stack (though if IFRAMEs are used, clicking the back button doesn't always result in what the user expects). However, requests made with XHR are *not* recorded in the browser's history. This too can pose an issue if your users are familiar with using the back button to navigate within your Web application.

The Usability Question

While we're talking about user expectations, we should mention usability. The Ajax approach is fairly new—there really aren't any established best practices or heuristics. However, standard Web design principles still apply. As time passes and more people experiment with this approach, we will find the limits and establish guidelines. That said, you should let your users guide you. Depending on how you choose to use Ajax in your application, you may be dynamically changing parts of your page; users who are accustomed to seeing the entire browser refresh may not notice that anything has changed. This issue has led to features such as the Yellow Fade Technique (YFT) popularized by 37signals, as used in the Ajax poster application Basecamp.

In a nutshell, YFT says, "Take the part of the page that changed, and make it yellow." Assuming yellow is not the dominant color of your application, the user is likely to notice this change. Over time, you fade the yellow color until it returns to the original background color. Obviously, you could choose any color you want; all you are doing is drawing attention to what has changed.

Perhaps YTF isn't right for your application; instead, you may choose to alert your users in a less obvious though no less useful manner. Gmail shows a red flashing "Loading" sign in the upper-right corner to indicate that it is fetching data (see Figure 1-7).

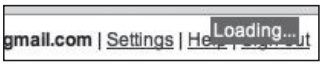

Figure 1-7. *Gmail's "Loading" sign*

Whether you need to use YFT or a similar technique is really up to your users. The simplest approach is to test it with a group of representative users. You could do this using paper-based or Web-based prototypes depending on where you are in the design process, but however you test it, you should get some user feedback before simply introducing complex Ajax usage.

You should also start small. Your first attempt at using Ajax shouldn't be to create a dynamic portal site with adjustable columns. Rather, begin by moving client-side validation to the server. Once you get the hang of it, begin to venture into more dynamic uses, such as populating a drop-down list or setting some default text.

However you choose to apply Ajax, remember to not do anything wacky. We know that this is not scientific advice. However, at this point, there aren't any hard-and-fast rules. Listen to your users, test before you deploy, and remember how quickly we all learned to hit the "skip intro" link.

You should be aware of a few common mistakes when using Ajax. We've already discussed providing visual clues to your users when something changes, but Ajax changes the standard Web approach in other ways. First, unlike IFRAMES and hidden frames, making a request via XHR does not modify the history stack of your browser. In many cases, this won't matter (how often have you clicked the back arrow only to discover that nothing has changed?), but if your users are expecting the back button to work, you will have some issues.

Unlike other browser-based approaches, Ajax won't modify the link displayed in the address bar, meaning you cannot easily bookmark a page or send a link to a friend. For many applications this may be desirable, but if your site provides driving directions or something similar, you will need a solution to this problem.

It's important to not go overboard with Ajax. Remember, JavaScript is running on your client's browser, and having thousands of lines of JavaScript can severely slow down your users' experiences. Poorly coded scripts can quickly spiral out of control, especially as volume increases.

Ajax allows you to perform operations asynchronously, which is both its greatest strength and one of its major weaknesses. We've trained users that our Web applications perform in a request/response paradigm, but with Ajax, we no longer have that limitation. We might be modifying a part of the page, and if the users aren't expecting that, they may be confused. So, you need to be careful to keep things obvious to your users—don't try to be too clever. Remember, when in doubt, test with representative users!

The Skill Set

If you've picked up this book, you probably have most of the skills necessary to implement Ajax in your application. Once again, we want to emphasize that Ajax is a client-side technique—it will work with whatever server-side technology you are currently using, whether it's Java, .NET, Ruby, PHP, or CGI. In fact, in this book, we will be pretty agnostic to the server side and assume you know best how to work with the server-side technology you use in your

day-to-day work. What we will focus on over the course of the next few hundred pages is the client-side techniques and technology you'll need to create rich browser-based applications.

While you can use any server-side technology you want, using Ajax involves a bit of a shift. In a typical Web application, server-side code renders an entire page and involves an entire unit of work. With Ajax, you might return only a small bit of text and might involve only a small subset of a business application. For most experienced Web developers, this won't be much of an issue, but it's something you need to keep in mind.

Emerging frameworks will help shield developers from some of the ins and outs of Ajax; however, you will need some familiarity with JavaScript. We know that JavaScript can be a real pain to use. Sorry, there's really no getting around it. Most of us learned to use "alerts" as a type of system out for debugging purposes, and sadly this technique is still widely used. However, we have new hammers in our toolkit.

Along with JavaScript, you will need some familiarity with other presentation-related technologies such as HTML, the DOM, and CSS. You don't need to be an expert, but you need to know the basics. We'll cover most of what you need to know in this book and supplement the text with various Web resources.

For the test driven among us (you do write unit tests, right?), we will cover JsUnit and Selenium (see Figure 1-8). These tools allow you to develop your JavaScript test first and check for browser compatibility testing. It is widely expected that the next generation of developer environments will feature improved support for JavaScript, and other Ajax-related technologies will further ease the burden on developers. Scripts and frameworks are being introduced regularly that also make development simpler.

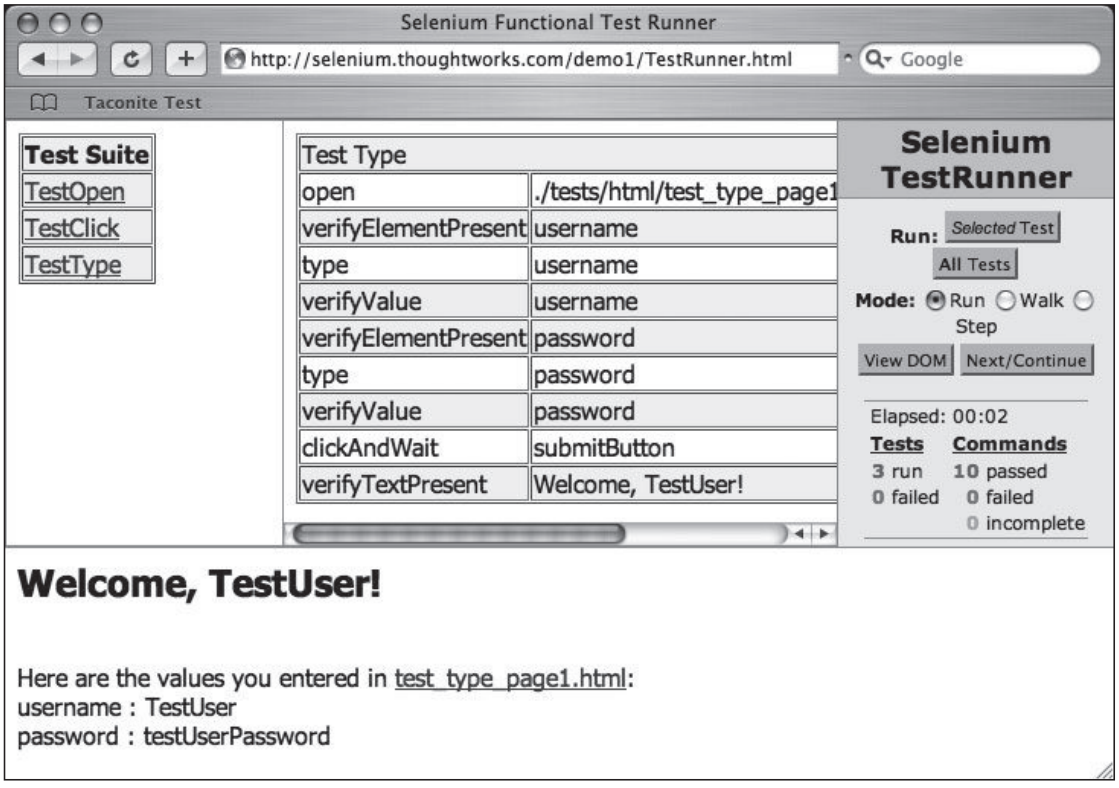

Figure 1-8. *Selenium*

Usage

Now that we've piqued your interest, it's important to know when you should and when you shouldn't use Ajax techniques. First, don't be afraid to try an approach on your applications. We believe almost every Web application can benefit from Ajax techniques; just don't go overboard until you have a good feel for where it makes sense. Validation is a perfect place to start, but don't limit yourself. Though you certainly can post data using Ajax, it probably shouldn't be the primary approach.

Second, browser issues are the only factor that should slow you down in your implementation of Ajax. If a large number of your users (or if particularly important users) are still using older browsers—anything before Internet Explorer 5, Safari 1.2, or Mozilla 1.0—these techniques won't work. If this is an important class of users, you will need to use cross-browser alternatives that work for your target audience, forgo Ajax altogether, or develop a site that will degrade gracefully. Browser support probably isn't a significant factor, because Netscape Navigator 4 usage is fractional; however, you should consult your Web logs to see what applies to your applications.

As we've stated, validation and form population are prime candidates for Ajax. You can also make truly dynamic portal sites using the DOM's drag technique shown in Google's personalized home page (see Figure 1-9).

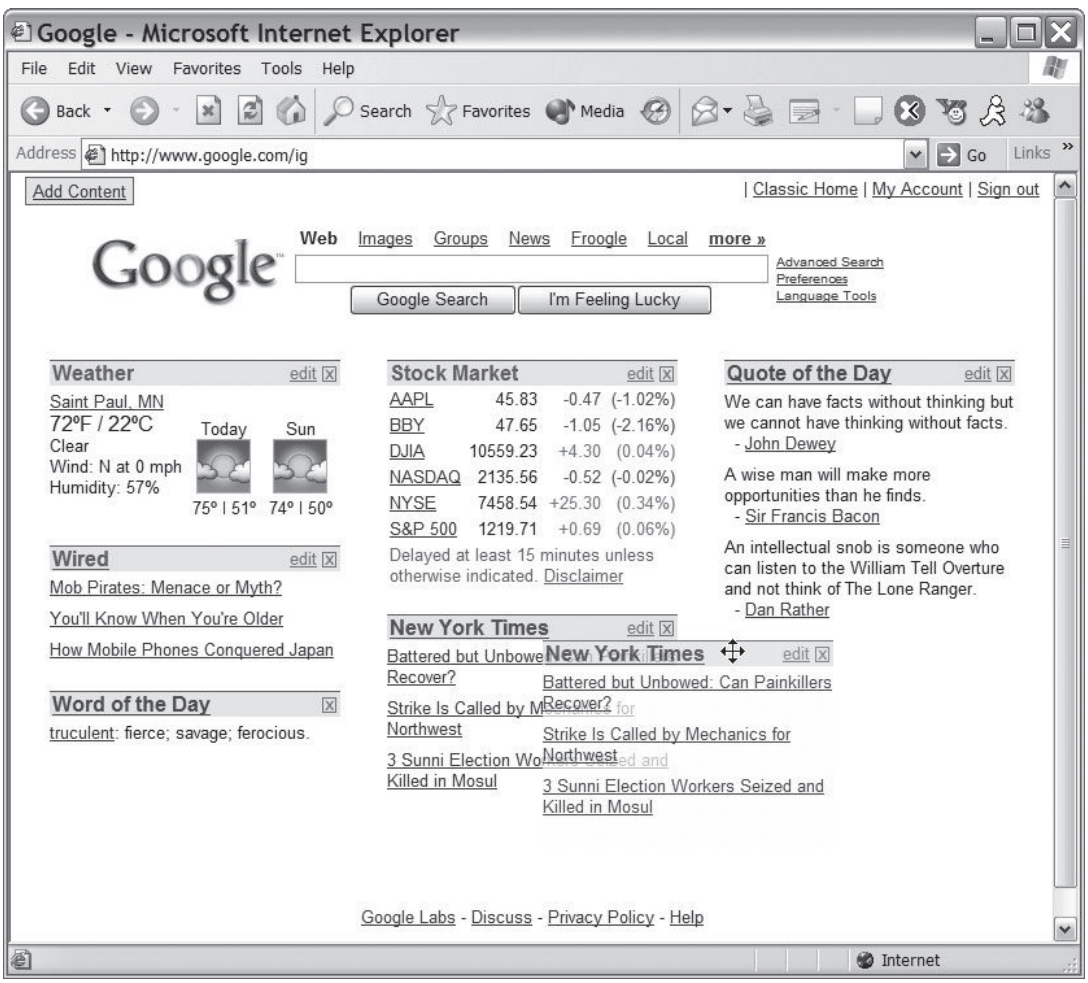

Figure 1-9. *Google's personalized home page*

As you can see, Ajax creates a whole new set of opportunities for Web application development. No longer are you hindered by proprietary techniques and disappointing compromises. With Ajax, the line between thick and thin is blurred, and the real winners are your users!

Design Considerations

Now that you have some ideas on where to use Ajax, we'll take a minute to cover some design considerations for applying Ajax. Many of these principles are no different from what you would normally do in a Web application, but they still bear mentioning. Strive to minimize the traffic between the client and server. Applied well, Ajax will make your application more responsive, but if you are trying to send an encyclopedia's worth of data back and forth every time your user navigates off a field, your users will not be amused. When in doubt, follow standard conventions. If most applications do XYZ, then you probably should too. When in doubt, look at the standards for Web desktop applications. Some patterns have been established, and more are sure to follow (`www.ajaxpatterns.org`).

When you first start using Ajax techniques, the way your application works may not be clear to your users. We've trained users over the years that Web applications work in a certain way, and Ajax adds an asynchronous component that may throw them. Put simply, don't surprise your users. If your pre-Ajax application didn't save the form when they tabbed off the last field, your post-Ajax application probably shouldn't either.

The most important issue to consider when implementing Ajax is simple: it's all about the user, stupid. Always remember the user, and don't practice "résumé-driven design." The desire to pass your next employer's buzzword filter is not a good enough reason to add Ajax to your application; if using Ajax benefits your user by providing a richer experience, by all means, get it out there. But don't forget—just because you can, doesn't mean you should. Be smart, think of your users first, and you should be fine.

We'll talk more about security later, but we want to point out that Ajax has some security considerations. Remember that source is viewable in a browser, meaning that anyone can figure out how you created your snappy widget. Since you have to include uniform resource locators (URLs) when you set up your XHR objects, it is possible that someone with malicious intent could hack your site and run their own code. Applying Ajax judiciously can mitigate this risk.

Summary

The Internet has certainly evolved from its early origins as a way for researchers to connect and share information. The Internet began with simple textual browsers and static pages, but it is now hard to find a company that doesn't have a polished Web site. In its early days, who could have possibly imagined that people would one day flock to the Internet to research that new car or buy the latest Stephen King novel?

Developers fed up with the difficulty of deploying thick client applications to thousands of users looked to the Web to ease their burden. Several Web application technologies have been developed over the years—some proprietary, others requiring significant programming abilities. Though some provided a richer user experience than others, no one would confuse a thin client application with its desktop-based cousin. Still, the ease of deployment, the ability to reach a wider customer base, and the lower cost of maintenance means that despite the limitations of browsers, they are still the target platform of choice for many applications.

Developers have used hacks to circumvent some of the most troublesome restrictions the Internet places on developers. Various remote scripting options and HTML elements let developers work asynchronously with the server, but it wasn't until the major browsers added support for the XMLHttpRequest object that a true cross-browser method was possible. With companies such as Google, Yahoo, and Amazon leading the way, we are finally seeing browser-based applications that rival thick clients. With Ajax, you get the best of both worlds—your code sits on a server that you control, and any customer with a browser can access an application that provides a full, rich user experience.

CHAPTER 2

Using the XMLHttpRequest Object

Now that we've discussed the history of dynamic Web applications and introduced Ajax, it's time to cover the heart of the matter: how to use the XMLHttpRequest object. While Ajax is more of a technique than a technology, without widespread support for XMLHttpRequest, Google Suggest and Ta-da List wouldn't exist as we currently know them. And you wouldn't be reading this book!

XMLHttpRequest was originally implemented in Internet Explorer 5 as an ActiveX component. That it worked only in Internet Explorer kept most developers from using XMLHttpRequest until its recent adoption as a de facto standard in Mozilla 1.0 and Safari 1.2. It's important to note that XMLHttpRequest is *not* a W3C standard, though much of the functionality is covered in a new proposal: the DOM Level 3 Load and Save Specification. Because it is not a standard, its behavior may differ slightly from browser to browser, though most methods and properties are widely supported. Currently, Firefox, Safari, Opera, Konqueror, and Internet Explorer all implement the behavior of the XMLHttpRequest object similarly.

That said, if a significant number of your users still access your site or application with older browsers, you will need to consider your options. As we discussed in Chapter 1, if you are going to use Ajax techniques, you need to either develop an alternative site or allow your application to degrade gracefully. With most usage statistics indicating that only a small fraction of browsers in use today lack XMLHttpRequest support, the chances of this being a problem are slim. However, you need to check your Web logs and determine what clients your customers are using to access your sites.

Overview of the XMLHttpRequest Object

You must first create an XMLHttpRequest object using JavaScript before you can use the object to send requests and process responses. Since XMLHttpRequest is not a W3C standard, you can use JavaScript in a couple of ways to create an instance of XMLHttpRequest. Internet Explorer implements XMLHttpRequest as an ActiveX object, and other browsers such as Firefox, Safari, and Opera implement it as a native JavaScript object. Because of these differences, the JavaScript code must contain logic to create an instance of XMLHttpRequest using the ActiveX technique or using the native JavaScript object technique.

The previous statement might send shivers down the spines of those who remember the days when the implementation of JavaScript and the DOM varied widely among browsers. Fortunately, in this case you don't need elaborate code to identify the browser type to know

how to create an instance of the XMLHttpRequest object. All you need to do is check the browser's support of ActiveX objects. If the browser supports ActiveX objects, then you create the XMLHttpRequest object using ActiveX. Otherwise, you create it using the native JavaScript object technique. Listing 2-1 demonstrates the simplicity of creating cross-browser JavaScript code that creates an instance of the XMLHttpRequest object.

Listing 2-1. *Creating an Instance of the XMLHttpRequest Object*

```
var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}
```

As you can see, creating the XMLHttpRequest object is rather trivial. First, you create a globally scoped variable named `xmlHttp` to hold the reference to the object. The `createXMLHttpRequest` method does the work of actually creating an instance of XMLHttpRequest. The method contains simple branching logic that determines how to go about creating the object. The call to `window.ActiveXObject` will return an object or `null`, which is evaluated by the `if` statement as true or false, thus indicating whether the browser supports ActiveX controls and thus is Internet Explorer. If so, then the XMLHttpRequest object is created by instantiating a new instance of ActiveXObject, passing a string indicating the type of ActiveX object you want to create. In this instance, you provide `Microsoft.XMLHTTP` to the constructor, indicating your desire to create an instance of XMLHttpRequest.

If the call to `window.ActiveXObject` fails, then the JavaScript branches to the `else` statement, which determines whether the browser implements XMLHttpRequest as a native JavaScript object. If `window.XMLHttpRequest` exists, then an instance of XMLHttpRequest is created.

Thanks to JavaScript's dynamically typed nature and that XMLHttpRequest implementations are compatible across various browsers, you can access the properties and methods of an instance of XMLHttpRequest identically, regardless of the method used to create the instance. This greatly simplifies the development process and keeps the JavaScript free of browser-specific logic.

Methods and Properties

Table 2-1 shows some typical methods on the XMLHttpRequest object. Don't worry; we'll talk about these methods in greater detail in a moment.

Table 2-1. *Standard XMLHttpRequest Operations*

Method	Description
`abort()`	The current request.
`getAllResponseHeaders()`	Returns all the response headers for the HTTP request as key/value pairs.
`getResponseHeader("header")`	Returns the string value of the specified header.
`open("method", "url")`	Sets the stage for a call to the server. The method argument can be either `GET`, `POST`, or `PUT`. The `url` argument can be relative or absolute. This method includes three optional arguments.
`send(content)`	Sends the request to the server.
`setRequestHeader("header", "value")`	Sets the specified header to the supplied value. `open()` must be called before attempting to set any headers.

Let's take a closer look at these methods.

`void open(string method, string url, boolean asynch, string username, string password)`: This method sets up your call to the server. This method is meant to be the script-only method of initializing a request. It has two required arguments and three optional arguments. You are required to supply the specific method you are invoking (`GET`, `POST`, or `PUT`) and the URL of the resource you are calling. You may optionally pass a Boolean indicating whether this call is meant to be asynchronous—the default is `true`, which means the request is asynchronous in nature. If you pass a `false`, processing waits until the response returns from the server. Since making calls asynchronously is one of the main benefits of using Ajax, setting this parameter to `false` somewhat defeats the purpose of using the XMLHttpRequest object. That said, you may find it useful in certain circumstances such as validating user input before allowing the page to be persisted. The last two parameters are self-explanatory, allowing you to include a specific username and password.

`void send(content)`: This method actually makes the request to the server. If the request was declared as asynchronous, this method returns immediately, otherwise it waits until the response is received. The optional argument can be an instance of a DOM object, an input stream, or a string. The content passed to this method is sent as part of the request body.

`void setRequestHeader(string header, string value)`: This method sets a value for a given header value in the HTTP request. It takes a string representing the header to set and a string representing the value to place in the header. Note that it must be called after a call to `open()`. Of all these methods, the two you will use the most are `open()` and `send()`. The XMLHttpRequest object has a number of properties that prove themselves quite useful while designing Ajax interactions.

`void abort()`: This method is really quite self-explanatory—it stops the request.

`string getAllResponseHeaders()`: The core functionality of this method should be familiar to Web application developers—it returns a string containing response headers from the HTTP request. Headers include `Content-Length`, `Date`, and `URI`.

`string getResponseHeader(string header)`: This method is a companion to `getAllResponseHeaders()` except it takes an argument representing the specific header value you want, returning this value as a string.

In addition to these standard methods, the XMLHttpRequest object exposes the properties listed in Table 2-2. You'll use these properties extensively when working with XMLHttpRequest.

Table 2-2. *Standard XMLHttpRequest Properties*

Property	Description
`onreadystatechange`	The event handler that fires at every state change, typically a call to a JavaScript function.
`readyState`	The state of the request. The five possible values are 0 = uninitialized, 1 = loading, 2 = loaded, 3 = interactive, and 4 = complete.
`responseText`	The response from the server as a string.
`responseXML`	The response from the server as XML. This object can be parsed and examined as a DOM object.
`status`	The HTTP status code from the server (that is, 200 for OK, 404 for Not Found, and so on).
`statusText`	The text version of the HTTP status code (that is, OK or Not Found, and so on).

An Example Interaction

At this point, you might be wondering what a typical Ajax interaction looks like. Figure 2-1 shows the standard interaction paradigm in an Ajax application.

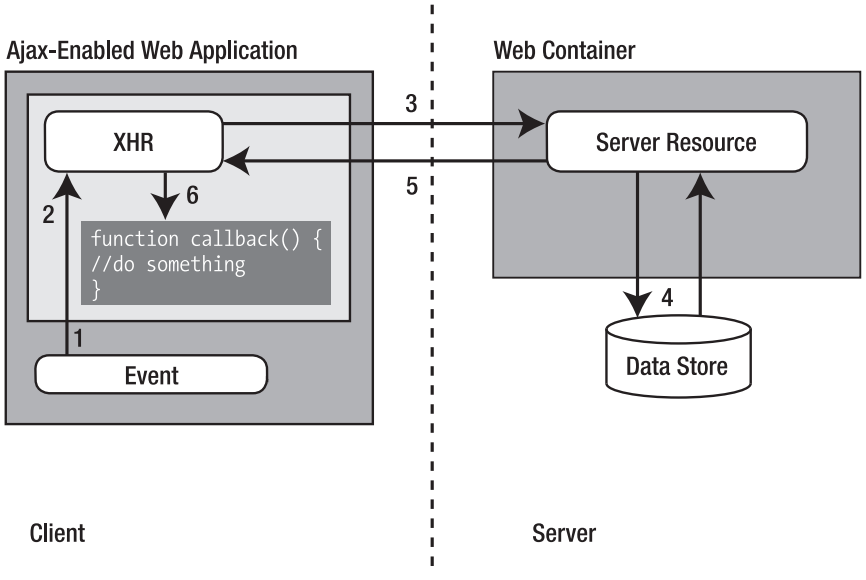

Figure 2-1. *Standard Ajax interaction*

Unlike the standard request/response approach found in a standard Web client, an Ajax application does things a little bit differently.

1. A client-side event triggers an Ajax event. Any number of things can trigger this, from a simple `onchange` event to some specific user action. You might have code like this:

   ```
   <input type="text"d="email" name="email" onblur="validateEmail()";>
   ```

2. An instance of the XMLHttpRequest object is created. Using the `open()` method, the call is set up—the URL is set along with the desired HTTP method, typically `GET` or `POST`. The request is actually triggered via a call to the `send()` method. This code might look something like this:

   ```
   var xmlHttp;
   function validateEmail() {
     var email = document.getElementById("email");
     var url = "validate?email=" + escape(email.value);
     if (window.ActiveXObject) {
       xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
     }
     else if (window.XMLHttpRequest) {
       xmlHttp = new XMLHttpRequest();
     }
   xmlHttp.open("GET", url);
   xmlHttp.onreadystatechange = callback;
   xmlHttp.send(null);
   }
   ```

3. A request is made to the server. This might be a call to a servlet, a CGI script, or any server-side technique.

4. The server can do anything you can think of, including accessing a data store or even another system.

5. The request is returned to the browser. The `Content-Type` is set to text/xml—the XMLHttpRequest object can process results only of the text/html type. In more complex instances, the response might be quite involved and include JavaScript, DOM manipulation, or other related technologies. Note that you also need to set the headers so that the browser will not cache the results locally. You do this with the following code:

   ```
   response.setHeader("Cache-Control", "no-cache");
   response.setHeader("Pragma", "no-cache");[1]
   ```

1. `Pragma` and `Cache-Control`…don't they do the same thing? Yes, they do, but `Pragma` is defined for backward compatibility.

6. In this example, you configure the XMLHttpRequest object to call the function `callback()` when the processing returns. This function checks the `readyState` property on the XMLHttpRequest object and then looks at the status code returned from the server. Provided everything is as expected, the `callback()` function does something interesting on the client. A typical callback method looks something like this:

```
function callback() {
  if (xmlHttp.readyState == 4) {
    if (xmlHttp.status == 200) {
        //do something interesting here
    }
  }
}
```

As you can see, this is different from the normal request/response paradigm but not in a way that is foreign to Web developers. Obviously, you have a bit more going on when you create and set up an XMLHttpRequest object and when the "callback" has some checks for states and statuses. Typically, you will wrap these standard calls into a library that you will use throughout your application, or you will use one that is available on the Web. This field is new, but a considerable amount of activity is happening in the open-source community.

In general, the various frameworks and toolkits available on the Web take care of the basic wiring and the browser abstractions, and some add user interface components. Some are purely client based; others require work on the server. Many of these frameworks have just begun development or are in the early phases of release; the landscape is constantly changing, with new libraries and versions coming out regularly. As the field matures, the best ones will become apparent. Some of the more mature libraries include libXmlRequest, RSLite, sarissa, JavaScript Object Notation (JSON), JSRS, Direct Web Remoting (DWR), and Ruby on Rails. This is a dynamic space, so keep your RSS aggregator tuned to those sites dedicated to posting about all things Ajax!

GET vs. POST

You might be wondering what the difference is between `GET` and `POST` and when you should use one or the other. In theory, use `GET` when the request is idempotent, meaning that multiple requests will return the same result. In truth, if your corresponding server method modifies state in some way, it's unlikely this is actually true. That said, it is the standard. The more practical difference comes in terms of payload size—in many cases, browsers and servers will limit the length of the URL used to send data to the server. In general, use `GET` to retrieve data from the server; in other words, avoid changing state on the server with a `GET` call.

In general, use `POST` methods anytime you are changing the state on the server. Unlike `GET`, you are required to set the `Content-Type` header on the XMLHttpRequest object like this:

```
xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Unlike `GET`, `POST` does not restrict the size of the payload that is sent to the server, and `POST` requests are not guaranteed to be idempotent.

Chances are, most of the requests you will make will be `GET` requests; however, `POST` is there if you need to use it.

Remote Scripting

Now that we've introduced Ajax, we want to briefly touch on remote scripting. Many of you might be thinking, "What's the big deal with Ajax? I've been doing that same thing with IFRAMEs for years." In fact, we have used this approach too. What we've done in the past is typically referred to as *remote scripting* and is thought of by many as a hack. However, it does provide a mechanism to avoid page refreshes.

Overview of Remote Scripting

Essentially, remote scripting is a type of remote procedure call. You are interacting with your server just like a normal Web application, but you don't refresh the entire page. Just as with Ajax, you can call any server-side technology that can receive requests, process those requests, and return a meaningful result. Just as on the server side, you have a number of options on the client side when implementing remote scripting. You can imbed a Flash animation, a Java applet, or an ActiveX component into your application. You can even go so far as using XML-RPC, but the complexity of this approach makes it less than ideal unless you are experienced with the techniques. Common implementations of remote scripting include combining scripting with an IFRAME (hidden or otherwise) and having the server return JavaScript, which is then run within the browser.

Microsoft has its own solution for remote scripting, cleverly called Microsoft Remote Scripting (MSRS), that allows you to call server scripts as if they were local. A Java applet is embedded in the page to facilitate the communication with the server, an `.asp` page is used to house the server-side scripts, and an `.htm` file manages the client side of the arrangement. You can use Microsoft's solution in Netscape and Internet Explorer versions 4.0 and greater. Calls can be made both synchronously and asynchronously. However, this solution requires Java, which may mean an additional installation routine, and it expects Internet Information Services (IIS), which limits your server-side options.

Brent Ashley has created two free cross-platform libraries for remote scripting. JSRS is a client-side JavaScript library that takes advantage of DHTML to make remote calls to the server. It works on a wide variety of operating systems and browsers. With common and popular server-side implementations (including PHP, Python, and Perl CGI), it is likely you can get JSRS up and running on your site. Ashley provides JSRS free of charge and makes the source code available on his Web site at `www.ashleyit.com/rs/main.htm`.

If JSRS is too heavyweight for your tastes, Ashley has also created RSLite, which uses cookies. It is limited to small amounts of data and single calls but is supported in most browsers.

A Remote Scripting Example

For comparison, we'll show you an example of how Ajax-like techniques are implemented using IFRAMEs. It's pretty straightforward, and we have used this approach in the past (before XMLHttpRequest came on the scene). This example won't actually call the server, but it gives you an idea of how to use IFRAMEs for remote scripting.

This example involves two files: `iframe.html` (see Listing 2-2) and `server.html` (see Listing 2-3). The `server.html` mimics the response that would actually be returned from a server.

Listing 2-2. *The* `iframe.html` *File*

```
<html>
  <head>
    <title>Example of remote scripting in an IFRAME</title>
  </head>
  <script type="text/javascript">
    function handleResponse() {
      alert('this function is called from server.html');
    }
  </script>
  <body>
  <h1>Remote Scripting with an IFRAME</h1>

  <iframe id="beforexhr"
  name="beforexhr"
  style="width:0px; height:0px; border: 0px"
  src="blank.html"></iframe>

  <a href="server.html" target="beforexhr">call the server</a>

  </body>
</html>
```

Listing 2-3. *The* `server.html` *File*

```
<html>
  <head>
    <title>the server</title>
  </head>
  <script type="text/javascript">
    window.parent.handleResponse();
  </script>
  <body>
  </body>
</html>
```

Figure 2-2 shows the original page. Running this code produces a result like the one shown in Figure 2-3.

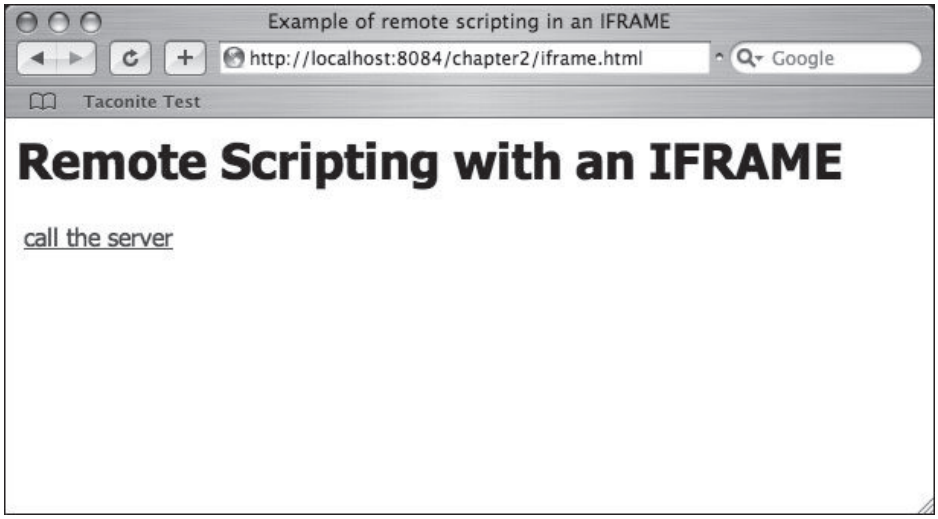

Figure 2-2. *The original page*

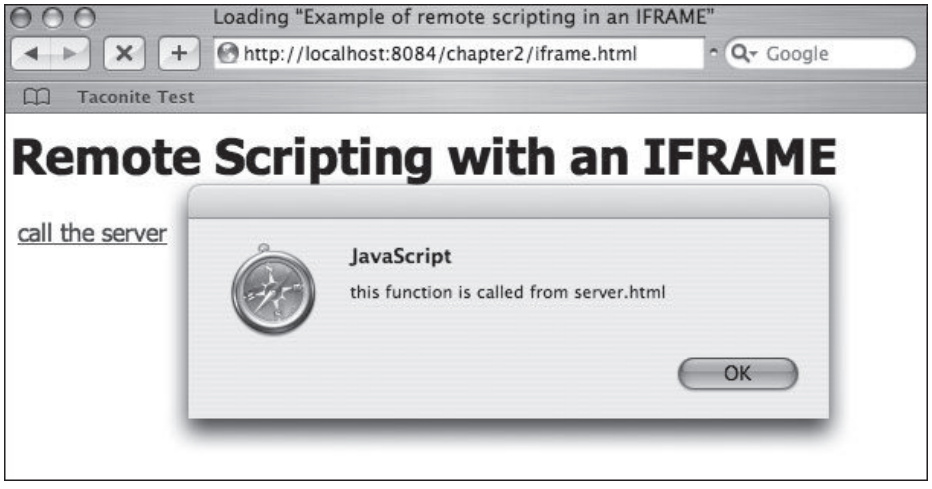

Figure 2-3. *The page after calling the "server"*

How to Send a Simple Request

You're now ready to start using the XMLHttpRequest object. We've just discussed how to create the object, so now we can show how to send requests to the server and process the server's response.

The simplest request you can make is the one where you don't send any information to the server in the form of query parameters or posted form data. In practice, you'll almost always want to send some information to the server.

The basic steps for sending a request using the XMLHttpRequest object are as follows:

1. Obtain a reference to an instance of XMLHttpRequest, either by creating a new one or accessing a variable that already holds an instance of XMLHttpRequest.

2. Tell the XMLHttpRequest object about the function that will handle changes in the XMLHttpRequest object's state. You accomplish this by setting the object's `onreadystatechange` property with a pointer to a JavaScript function.

3. Assign the properties of the request. The XMLHttpRequest object's `open()` method assigns the impending request. The `open()` method takes three parameters: a string indicating the method (usually either `GET` or `POST`), a string representing the URL of the destination resource, and a Boolean indicating whether the request should be made asynchronously.

4. Send the request to the server. The XMLHttpRequest object's `send()` method transmits the request to the indicated destination resource. The `send()` method accepts one parameter, which is typically a string or a DOM object. This parameter is transmitted to the destination URL as part of the request body. When providing a parameter to the `send()` method, be sure the method assigned in the `open()` method is `POST`. Use `null` when there is no data to be sent as part of the request body.

These steps are intuitive: you need an instance of the XMLHttpRequest object, you need to tell it what to do when it undergoes changes in state, you need to tell it where and how to send the request, and finally you need to direct XMLHttpRequest to transmit the request. However, unless you're experienced with C or C++, the notion of a *function pointer* may be foreign to you.

A function pointer is just like any variable, except that instead of pointing to data like a string or number or even an instance of an object, it points to a function. In JavaScript, all functions are addressed in memory and can be referenced using the function name. This gives you the flexibility of passing function pointers as parameters to other functions or storing a function pointer in an object's properties.

In the case of the XMLHttpRequest object, the `onreadystatechange` property stores the pointer to the callback function. The callback function is called when the XMLHttpRequest object's internal state changes. When an asynchronous call is made, the request is transmitted, and the script continues processing immediately—the script doesn't have to wait for the request to complete before continuing. Once the request is sent, the object's `readyState` property undergoes numerous changes. While you could base processing on any state, typically you are most interested in the one that indicates the server is finished sending the response. By setting the callback function, you are effectively telling the XMLHttpRequest object, "Call this function to handle the response whenever it arrives."

A Simple Request Example

This first example is simple. It's a small HTML page with a single button. Clicking the button initiates an asynchronous request to the server. The server will respond by sending a simple static text file. The response is handled by displaying the contents of the static text file in an alert window. Listing 2-4 shows the HTML page and associated JavaScript.

Listing 2-4. *The* `simpleRequest.html` *Page*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Simple XMLHttpRequest</title>
```

```
<script type="text/javascript">
var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}

function startRequest() {
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET", "simpleResponse.xml", true);
    xmlHttp.send(null);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            alert("The server replied with: " + xmlHttp.responseText);
        }
    }
}
</script>
</head>

<body>
    <form action="#">
        <input type="button" value="Start Basic Asynchronous Request"
                onclick="startRequest();"/>
    </form>
</body>
</html>
```

The server's response file, `simpleResponse.xml`, contains only a single line of text. Clicking the button on the HTML page should produce an alert box with the contents of the `simpleResponse.xml` file. Figure 2-4 displays the identical alert boxes that contain the server's response in both Internet Explorer and Firefox.

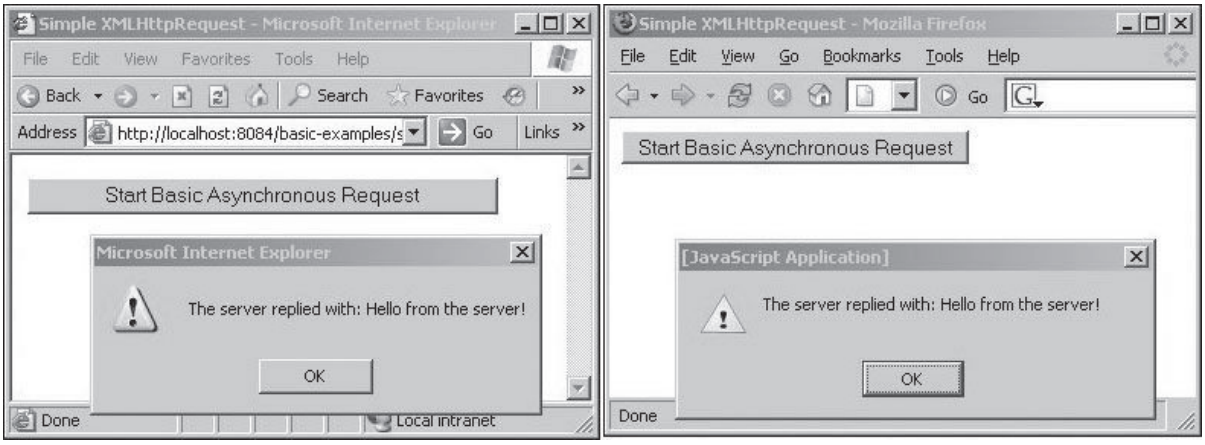

Figure 2-4. *The first simple asynchronous request*

The request to the server was sent asynchronously, allowing the browser to continue responding to user input while awaiting the server's response in the background. If a synchronous operation was chosen and if the server's response had taken several seconds to arrive, the browser would have been unresponsive to user input during the waiting time. The asynchronous behavior, while subtle, can measurably improve the end user's experience by avoiding the appearance that the browser has frozen and is failing to respond to user input. This allows the user to continue working while the server works on the previous request in the background.

The ability to communicate with the server without interrupting the user's workflow opens a wide variety of techniques for improving the user experience. One application, for example, is for validating user input. While a user is filling out the fields on an input form, the browser could periodically send the form values to the server for validation, without interrupting the user filling in the remaining form fields. If a validation rule fails, the user could be notified immediately, before the form is actually sent to the server for processing, saving the user time and reducing the load on the server, as the form's contents don't have to be rebuilt after an unsuccessful form submission.

A Word About Security

Any discussion of browser-based technologies wouldn't be complete without mentioning security. The XMLHttpRequest object is subjected to the browser's security "sandbox." Any resources requested by the XMLHttpRequest object must reside within the same domain from which the calling script originated. This security restriction prevents the XMLHttpRequest object from requesting resources outside the domain from which the script was originally served.

The strength of this security restriction varies by browser (see Figure 2-5). Internet Explorer shows an alert stating that a potential security risk exists but gives the user a choice of whether to continue with the request. Firefox simply stops the request and shows an error message in the JavaScript console.

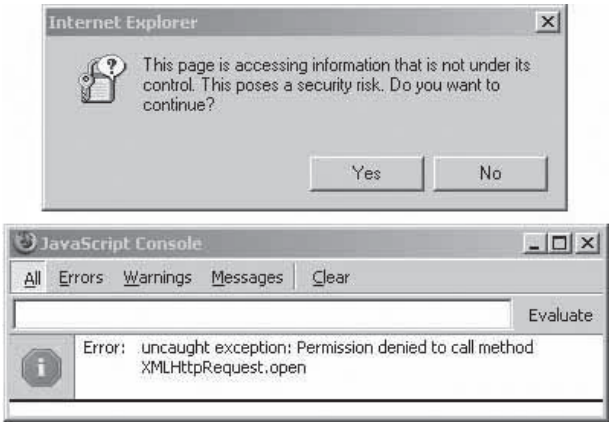

Figure 2-5. *Internet Explorer's and Firefox's responses to potential security threats*

Firefox does provide some JavaScript tricks to allow XMLHttpRequest to request resources from an outside URL. However, since these techniques are specific to the browser, it's best to not use them and to avoid attempting to access outside URLs using XMLHttpRequest.

DOM Level 3 Load and Save

So far, none of the solutions we've discussed is a standard. Though XMLHttpRequest enjoys widespread support, you've already seen how the process of creating it differs from browser to browser. Many people mistakenly believe Ajax has W3C backing; however, it does not. The W3C has addressed this and other shortcomings in a new standard with a rather lengthy name: the DOM Level 3 Load and Save Specification. This specification is designed to be a platform- and language-independent way to modify the content of a DOM document with XML content. The 1.0 version was recommended in April 2004, but currently no browser has implemented it.

When will Load and Save replace Ajax? Your bet is as good as ours. Considering how many browsers don't fully support existing standards, it's hard to say, but as more sites and applications take advantage of Ajax techniques, expect to see support in future releases. However, based on the length of time it took for earlier DOM versions to be adopted, don't hold your breath. In an interview, DOM Activity Lead Philippe Le Hégaret said it will take "significant time" to see widespread adoption. Some support exists for DOM Level 3—Opera's XMLHttpRequest implementation is based on its work in supporting DOM Level 3, and the Java API for XML Processing (JAXP) version 1.3 supports it. However, the emergence of a W3C specification signals the importance of Ajax techniques.

Load and Save is the culmination of an effort that began in August 1997 as a way of solving the incompatibilities in the browsers of the day. You may notice that the title says *Level 3*—what were Levels 1 and 2 all about? Level 1 was finished in October 1998 and gave us HTML 4.0 and XML 1.0. Today, Level 1 is widely supported. In November 2000, Level 2 was completed, though its adoption has been slow. CSS was part of Level 2.

What will developers gain from the Load and Save specification? Ideally, it will solve many of the cross-browser issues we currently encounter. Though Ajax is pretty straightforward, you'll recall that just to create an instance of the XMLHttpRequest object you need to check the browser type. A true W3C specification would alleviate those types of coding hacks. Basically,

Load and Save will provide Web developers with a common API to access and modify the DOM in a language- and platform-independent way. In other words, it won't matter if you're on Windows or Linux, and it won't matter if you're developing in VBScript or JavaScript. You will also be able to save a DOM tree as an XML document or take an XML document and load it into the DOM. The specification also provides support for XML 1.1, XML Schema 1.0, and SOAP 1.2. Once available, it is expected to be widely used by developers.

The DOM

We keep talking about the DOM, and if you haven't done a lot of work on the client side, you might not know what the DOM is. The DOM is a W3C specification for a platform- and language-independent way of accessing and modifying the content and structure of a document. In other words, it's a common way to represent and manipulate an HTML or XML document.

It's important to note that the design of the DOM is based on specifications from the Object Management Group, which allows it to be used with any programming language. It was originally conceived as a way of making JavaScript portable across browsers, though it has expanded beyond that limited application.

The DOM really is an object model in the object-oriented sense. The DOM defines the objects needed to represent and modify documents, the behavior and attributes of these objects, and the relationship between these objects. You can think of the DOM as a tree representation of the data and structure on a page, though of course it may not actually be implemented that way. Say you have a Web page that looks something like Listing 2-5.

Listing 2-5. *Simple Table*

```
<table>
  <tbody>
    <tr>
       <td>Foo</td>
       <td>Bar</td>
    </tr>
  </tbody>
</table>
```

You can picture the DOM of this simple table as something like Figure 2-6.

The beauty of the DOM specification is that it gives you a standard way to interact with your documents. Without the DOM, the most interesting aspects of Ajax wouldn't be possible. Since the DOM allows you to not only traverse but also edit the content, you can make highly dynamic pages.

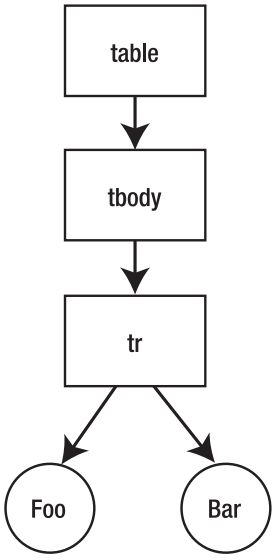

Figure 2-6. *Simple DOM*

Summary

While Ajaxesque techniques have been used for many years, the recent adoption of the XMLHttpRequest object by modern browsers has ushered in a new era of developing rich Web applications. In this chapter, we established the basics of working with the heart of Ajax, the XMLHttpRequest object. At this point, you know the methods and properties of the XMLHttpRequest object, and we've shown you some simple examples of their use. As you can see, the object is pretty straightforward and hides much of its complexity from you. Combined with a healthy dose of JavaScript and some basic DOM manipulation, Ajax allows for a level of interactivity previously unmatched on the Web.

As mentioned in Chapter 1, with XMLHttpRequest you are no longer limited to complete page refreshes and synchronous conversations with your server. In the upcoming chapters, we'll show you how to combine your existing expertise of server-side technologies with the unique capabilities of XMLHttpRequest to provide highly interactive Web applications.

CHAPTER 3

Communicating with the Server: Sending Requests and Processing Responses

Now the real fun begins! It's time to put your newfound knowledge of the XMLHttpRequest object to work. By showing some simple examples, we'll demonstrate how the XMLHttpRequest object sends requests to the server and how to process the server response with JavaScript.

Note The examples in this chapter do not use a dynamic server to process the response and provide a real-time response. Instead, the examples use simple text files to mimic the server's response. Doing so removes complexity, allowing you to focus on what's happening in the browser.

Processing the Server Response

The XMLHttpRequest object provides two properties that provide access to the server response. The first property, `responseText`, simply provides the response as a string. The second property, `responseXML`, provides the response as an XML object. Retrieving the response as simple text is fine for simple use cases, such as when the response is displayed in an alert box or the response is a simple one-word phrase indicating success or failure.

The previous example in Chapter 2 accessed the server response using the `responseText` property and displayed it in an alert box.

Using the innerHTML Property to Create Dynamic Content

Accessing the server response as simple text doesn't provide much flexibility. Simple text lacks structure and is difficult to parse logically with JavaScript. It also makes it more difficult to dynamically generate page content.

The `responseText` property becomes useful if you utilize it in conjunction with the `innerHTML` property of HTML elements. The `innerHTML` property is a nonstandard property first implemented by Internet Explorer and later by many other popular browsers. It is a simple string that represents the content between a set of start and end tags.

By using `responseText` and `innerHTML` together, the server can produce HTML content that is consumed by the browser using the `innerHTML` property. The following example illustrates search functionality using the XMLHttpRequest object, its `responseText` property, and the `innerHTML` property of HTML elements. Clicking the search button initiates a "search" on the server. The server responds by generating a table of results. The browser processes the response by setting the `innerHTML` property of a `div` element to the value of the XMLHttpRequest object's `responseText` property. Figure 3-1 shows the browser window after the search button has been clicked and the results table has been added to the window content.

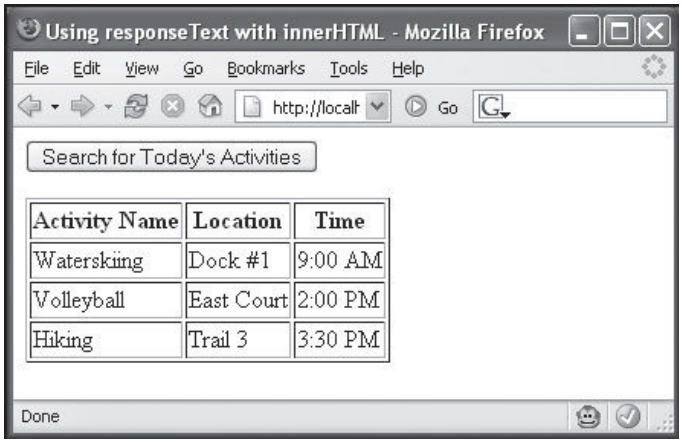

Figure 3-1. *The browser window showing the search results retrieved using XMLHttpRequest and processed using* `innerHTML`

The code for this example is similar to the example from Chapter 2 that simply displayed the server response in an alert box. The steps are as follows:

1. Clicking the search button calls the `startRequest` function, which first calls the `createXMLHttpRequest` function to initialize a new instance of the XMLHttpRequest object.
2. The `startRequest` function sets the callback function to be the `handleStateChange` function.
3. The `startRequest` function continues by using the `open()` method to set the request's method (`GET`) and its destination and to perform the request asynchronously.
4. The request is then sent using the `send()` method of the XMLHttpRequest object.
5. Each time the XMLHttpRequest object's internal state changes, the `handleStateChange` function is called. Once the response has been received (indicated by the `readyState` property having a value of 4), the `innerHTML` property of the `div` element is set using the `responseText` property of XMLHttpRequest.

Listing 3-1 shows `innerHTML.html`, and Listing 3-2 shows `innerHTML.xml`, which represents the content produced by the search.

Listing 3-1. innerHTML.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Using responseText with innerHTML</title>

<script type="text/javascript">
var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}

function startRequest() {
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET", "innerHTML.xml", true);
    xmlHttp.send(null);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            document.getElementById("results").innerHTML = xmlHttp.responseText;
        }
    }
}
</script>
</head>

<body>
    <form action="#">
        <input type="button" value="Search for Today's Activities"
                onclick="startRequest();"/>
    </form>
    <div id="results"></div>
</body>
</html>
```

Listing 3-2. innerHTML.xml

```
<table border="1">
    <tbody>
        <tr>
            <th>Activity Name</th>
            <th>Location</th>
            <th>Time</th>
        </tr>
        <tr>
            <td>Waterskiing</td>
            <td>Dock #1</td>
            <td>9:00 AM</td>
        </tr>
        <tr>
            <td>Volleyball</td>
            <td>East Court</td>
            <td>2:00 PM</td>
        </tr>
        <tr>
            <td>Hiking</td>
            <td>Trail 3</td>
            <td>3:30 PM</td>
        </tr>
    </tbody>
</table>
```

Using `responseText` and `innerHTML` greatly simplifies adding dynamic content to the page. Unfortunately, the approach has its drawbacks. As mentioned, the `innerHTML` property is not a standard property of HTML elements, making its implementation by standards-compliant browsers optional. Most modern browsers, however, support the `innerHTML` property. Ironically, Internet Explorer, the browser that pioneered the use of `innerHTML`, has the most limited implementation of it. Many of today's modern browsers expose the `innerHTML` property as a read/write property on all HTML elements. Internet Explorer, on the other hand, restricts the `innerHTML` property to read-only on HTML elements such as tables and table rows, somewhat limiting its usefulness.

Parsing the Response As XML

You have already seen that the server doesn't necessarily need to send the response in XML format. It's perfectly legal to send the response as simple text as long as the `Content-Type` response header is set to text/plain, as opposed to text/xml for XML. Complex data structures are good candidates to be sent in XML format. Modern browsers have consistently good support for navigating an XML document and also for modifying the XML document's structure and contents.

How exactly does the browser handle XML returned by the server? Modern browsers treat the XML as an XML document in accordance with the W3C DOM. The W3C DOM specifies a rich set of APIs for searching and manipulating XML documents. DOM-compliant browsers

are required to implement these APIs and exhibit the prescribed behavior, maximizing the portability of scripts between browsers.

The W3C Document Object Model

What exactly is the W3C DOM? The W3C Web page provides this clear definition:

> *The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure, and style of documents. The document can be further processed, and the results of that processing can be incorporated back into the presented page.*

Furthermore, the W3C explains the motivation behind defining a standard DOM. The W3C received numerous requests from its members regarding the method by which the object model of XML and HTML documents should be exposed to scripts. The submissions do not propose any new tags or style sheet technology but rather attempt to ensure that interoperable and scripting language–neutral solutions are agreed upon and embraced by the community. In short, the W3C DOM standard intends to help avoid the scripting nightmares of the late 1990s when competing browsers had their own proprietary, often incompatible object models, which made cross-platform scripting exceedingly difficult.

The W3C DOM and JavaScript

It's easy to confuse the W3C DOM with JavaScript. The DOM is an API for HTML and XML documents that provides a structural representation of the document and defines how the document structure is accessed through script. JavaScript is a language that is *used* to access and manipulate the DOM. Without the DOM, JavaScript would not have any notion of Web pages and the elements that make up the pages. Every element within the document is part of the DOM, making the element's properties and methods available to JavaScript.

The DOM is independent of any programming language. Typically the DOM is accessed through JavaScript, although this is not a requirement. You can use any scripting language to access the DOM, thanks to its single, consistent API. Table 3-1 lists useful properties of DOM elements, and Table 3-2 lists useful methods.

Table 3-1. *Properties of DOM Elements Useful for Processing XML Documents*

Property Name	Description
`childNodes`	Returns an array of the current element's children
`firstChild`	Returns the first direct child of the current element
`lastChild`	Returns the last child of the current element
`nextSibling`	Returns the element immediately following the current element
`nodeValue`	Specifies the read/write property representing the element's value
`parentNode`	Returns the element's parent node
`previousSibling`	Returns the element immediately preceding the current element

Table 3-2. *Methods of DOM Elements Useful for Traversing XML Documents*

Method Name	Description
`getElementById(id) (document)`	Retrieves the element in the document that has the specified unique ID attribute value
`getElementsByTagName(name)`	Returns an array of the current element's children that have the specified tag name
`hasChildNodes()`	Returns a Boolean indicating whether the element has any child elements
`getAttribute(name)`	Returns the value of the element's attribute specified by `name`

Thanks to the W3C DOM, you can harness the power and flexibility of XML as the communication medium between the browser and server by writing simple, cross-browser scripts.

The following example demonstrates how easily you can read an XML document using W3C DOM-compliant JavaScript. Listing 3-3 shows the content of the XML document returned by the server to the browser. It's a simple list of U.S. states where the states are broken down by region.

Listing 3-3. *List of U.S. States Returned by the Server*

```
<?xml version="1.0" encoding="UTF-8"?>
<states>
    <north>
        <state>Minnesota</state>
        <state>Iowa</state>
        <state>North Dakota</state>
    </north>
    <south>
        <state>Texas</state>
        <state>Oklahoma</state>
        <state>Louisiana</state>
    </south>
    <east>
        <state>New York</state>
        <state>North Carolina</state>
        <state>Massachusetts</state>
    </east>
    <west>
        <state>California</state>
        <state>Oregon</state>
        <state>Nevada</state>
    </west>
</states>
```

In the browser, this yields a simple HTML page with two buttons. Clicking the first button loads the XML document from the server and then displays in an alert box all the states listed in the document. Clicking the second button loads the XML document from the server and displays just the northern states in an alert box (see Figure 3-2).

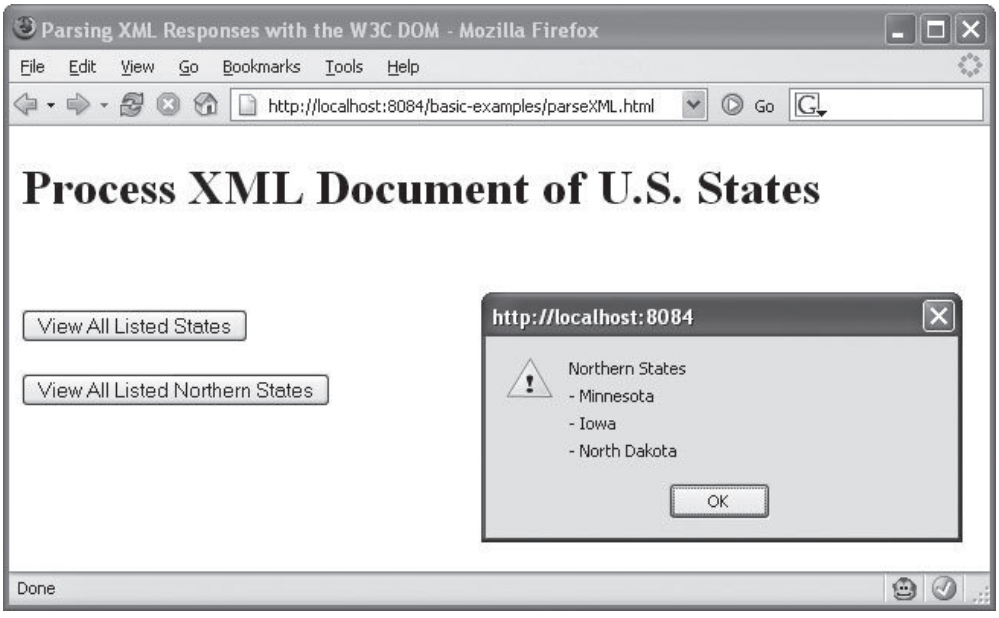

Figure 3-2. *Clicking either of the buttons on the page loads the XML document from the server and displays the appropriate results in an alert box.*

Listing 3-4 shows `parseXML.html`.

Listing 3-4. `parseXML.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Parsing XML Responses with the W3C DOM</title>

<script type="text/javascript">
var xmlHttp;
var requestType = "";

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}
```

```
function startRequest(requestedList) {
    requestType = requestedList;
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET", "parseXML.xml", true);
    xmlHttp.send(null);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            if(requestType == "north") {
                listNorthStates();
            }
            else if(requestType == "all") {
                listAllStates();
            }
        }
    }
}

function listNorthStates() {
    var xmlDoc = xmlHttp.responseXML;
    var northNode = xmlDoc.getElementsByTagName("north")[0];

    var out = "Northern States";
    var northStates = northNode.getElementsByTagName("state");

    outputList("Northern States", northStates);
}

function listAllStates() {
    var xmlDoc = xmlHttp.responseXML;
    var allStates = xmlDoc.getElementsByTagName("state");

    outputList("All States in Document", allStates);
}

function outputList(title, states) {
    var out = title;
    var currentState = null;
    for(var i = 0; i < states.length; i++) {
        currentState = states[i];
        out = out + "\n- " + currentState.childNodes[0].nodeValue;
    }
```

```
    alert(out);
}
</script>
</head>

<body>
    <h1>Process XML Document of U.S. States</h1>
    <br/><br/>
    <form action="#">
        <input type="button" value="View All Listed States"
               onclick="startRequest('all');"/>
        <br/><br/>
        <input type="button" value="View All Listed Northern States"
                 onclick="startRequest('north');"/>
    </form>
</body>
</html>
```

The scripting to retrieve the XML document from the server and process it is similar to the examples you saw previously when processing the response as simple text. The key differences lie in the `listNorthStates` and `listAllStates` functions. The previous examples retrieved the server response from the XMLHttpRequest object as text by using the object's `responseText` property. The `listNorthStates` and `listAllStates` functions instead use the XMLHttpRequest object's `responseXML` property to retrieve the results as an XML document, and doing so allows you to use the W3C DOM methods to traverse the XML document.

Study the `listAllStates` function. The first thing it does is create a local variable named `xmlDoc` and initialize it to the XML document returned by the server using the XMLHttpRequest object's `responseXML` property. You utilize the `getElementsByTagName` method of the XML document to retrieve all the elements in the document that have a tag name of `state`. The `getElementsByTagName` method returns an array of all the `state` elements that is assigned to the local variable named `allStates`.

After retrieving all the `state` elements from the XML document, the `listAllStates` function calls the `outputList` function to display the `state` elements in an alert box. The `listAllStates` method iterates over the array of `state` elements and for each element appends the state name to the string that is eventually displayed in the alert box.

One particular item to note is *how* the state name is retrieved from the `state` element. You might expect that the `state` element simply provides a property or method to retrieve the element's text. This is not the case.

The text representing the state's name is actually a child element of the `state` element. In an XML document, text is considered a node unto itself and must be the child of some other element. Since the text representing the state's name is actually a child element of the `state` element, it must be retrieved by first retrieving the text element from the `state` element and then asking the text element for its textual content.

The `outputList` function does just that. It iterates through all the elements in the array, assigning the current element to the `currentState` variable. Because the text element representing the state name is always the first child of the `state` element, you use the `childNodes` property to retrieve the text element. Once you have the actual text element, the `nodeValue` property returns the text content representing the state's name.

The `listNorthStates` function is similar to `listAllStates` except with an added twist. You want only the states that are in the northern region, not all the states. To do this, you first retrieve the `north` tag using the `getElementsByTagName` method to retrieve the `north` element from the XML document. Because the document contains only one `north` element and because the `getElementsByTagName` method always returns an array, you use the `[0]` notation to extract the `state` element since it's in the first (and only) position in the array returned by the `getElementsByTagName` method. Now that you have the `north` element, you can retrieve the `state` elements that are children of the `north` element by calling the `getElementsByTagName` method on the `north` element. Once you have an array of all the `state` elements that are children of the `north` element, you can again use the `outputList` method to display the states in an alert box.

Dynamically Editing Page Content with the W3C DOM

The Web has evolved from a medium for distributing static text documents to an application development platform in its own right. Legacy enterprise systems often deployed through text-only terminals or as client-server applications are being replaced by systems that are deployed completely through a Web browser.

As end users become more accustomed to using Web-based applications, they are beginning to demand a richer user experience. No longer are they satisfied with a complete page refresh every time they edit some data on the page. They want to see the results *right now* without waiting for an entire round-trip to the server.

You've now seen how simple it is to parse XML messages sent by the server. The W3C DOM provides properties and methods that allow you to traverse the XML structure and extract the desired data.

The previous example didn't do anything useful with the XML response sent by the server. Displaying the values of the XML document in an alert box doesn't provide much real-world value. What you really want to do is bring a rich client feel to the user by avoiding the constant page refreshing found in typical Web applications. Not only is the constant page refreshing annoying to the user, but also it wastes precious processor cycles on the server by rebuilding the entire page's content and needlessly using network bandwidth to carry the refreshed page.

The best solution, of course, is to change the existing content on the page on an as-needed basis. Instead of refreshing the entire page when most of the data on the page probably hasn't changed, we prefer to simply change the parts of the page where information has changed.

Traditionally this has been difficult to do within the confines of the Web browser. The browser is really just a tool that interprets special markup tags (HTML) and displays them according to a set of predefined rules. The Web, and thus Web browsers, was originally intended to display static information—information that didn't change without requesting new data from the server in the form of a new page.

With some exceptions, modern browsers represent the contents of a Web page using the W3C DOM. Doing so ensures that Web pages will be rendered identically among different browsers and that scripts intended for modifying the contents of the page will behave identically across browsers. The continuing maturity of the W3C DOM and JavaScript implementations by Web browsers has greatly simplified the task of creating content dynamically on the browser. Extensive hacks to work around the incompatibilities among browsers have largely evaporated. Table 3-3 lists the useful DOM properties and methods for creating content dynamically.

Table 3-3. *W3C DOM Properties and Methods Useful When Creating Content Dynamically*

Property/Method	Description
`document.createElement(tagName)`	The `createElement` method on the document object creates the element specified by `tagName`. Providing the string `div` as the method parameter produces a `div` element.
`document.createTextNode(text)`	This document object's `createTextNode` method creates a node containing static text.
`<element>.appendChild(childNode)`	The `appendChild` method adds the specified node to the current element's list of child nodes. For example, you can add an `option` element as a child node of a `select` element.
`<element>.getAttribute(name)` `<element>.setAttribute(name, value)`	These methods, respectively, get and set the value of the attribute `name` of the element.
`<element>.insertBefore(newNode, targetNode)`	This inserts the node `newNode` before the element `targetNode` as a child of the current element.
`<element>.removeAttribute(name)`	This removes the attribute `name` from the element.
`<element>.removeChild(childNode)`	This removes the element `childNode` from the element.
`<element>.replaceChild(newNode, oldNode)`	This method replaces the node `oldNode` with the node `newNode`.
`<element>.hasChildnodes()`	This method returns a Boolean indicating whether the element has any child elements.

A WORD ABOUT BROWSER INCOMPATIBILITIES

Despite the continually improving implementation of the W3C DOM and JavaScript in modern Web browsers, some quirks and incompatibilities still cause headaches when developing with the DOM and JavaScript.

Internet Explorer has the most limited implementation of the W3C DOM and JavaScript. In the early 2000s, Internet Explorer by some accounts held more than 95 percent of the total browser market, and with no competition in sight, Microsoft chose not to completely implement the various Web standards.

You can work around most of these quirks, although doing so makes the scripting messier and non-standard. For example, a `<tr>` element added directly to a `<table>` using `appendChild` will not appear in Internet Explorer, but it does in other browsers. The workaround is to add the `<tr>` element to the table's `<tbody>` element. This workaround performs correctly in all browsers.

Internet Explorer also has trouble with the `setAttribute` method. Internet Explorer won't correctly set the `class` attribute using `setAttribute`. The cross-browser workaround is to use both `setAttribute("class", "newClassName")` and `setAttribute("className", "newClassName")`. Also, you cannot set the `style` attribute using `setAttribute` in Internet Explorer. Instead of using `<element>.setAttribute("style, "font-weight:bold;")`, the most browser-compatible technique is `<element>.style.cssText = "font-weight:bold;";`.

The examples in this book will adhere as closely as possible to the W3C DOM and JavaScript standards but will stray from the standards when necessary to ensure compatibility with most modern browsers.

The following example demonstrates how you can use the W3C DOM and JavaScript to dynamically create content. The example is a fictional search engine for real estate listings. Clicking the Search button on the form will retrieve the results in XML format using the XMLHttpRequest object. The response XML will be processed using JavaScript to produce a table that lists the results of the search (see Figure 3-3).

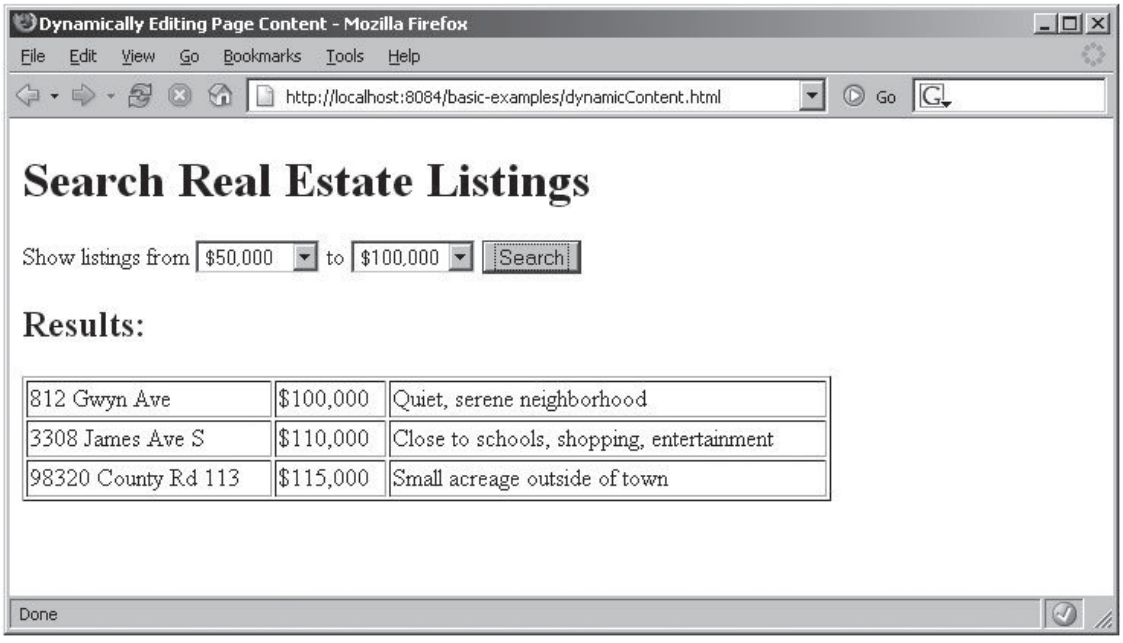

Figure 3-3. *The results of the search were created dynamically using W3C DOM methods and JavaScript.*

The XML returned by the server is simple (see Listing 3-5). The root `properties` node contains all the resulting property elements. Each `property` element contains three child elements: `address`, `price`, and `comments`.

Listing 3-5. `dynamicContent.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
    <property>
        <address>812 Gwyn Ave</address>
        <price>$100,000</price>
        <comments>Quiet, serene neighborhood</comments>
    </property>
    <property>
        <address>3308 James Ave S</address>
        <price>$110,000</price>
        <comments>Close to schools, shopping, entertainment</comments>
    </property>
```

```
    <property>
        <address>98320 County Rd 113</address>
        <price>$115,000</price>
        <comments>Small acreage outside of town</comments>
    </property>
</properties>
```

The JavaScript for actually sending the request to the server and responding to its response is the same as previous examples. The differences begin in the `handleReadyStateChange` function. Assuming the request completes successfully, the first thing that happens is that the content created by any previous searches is deleted by calling the `clearPreviousResults` function.

The `clearPreviousResults` function performs two tasks: removing the "Results" header text that appears at the top and clearing any rows from the results table. The header text is removed by first checking to see whether the `span` that surrounds the header text has any children by using the `hasChildNodes` method. You know that the header text exists if the `hasChildNodes` method returns `true`; if it does, delete the first (and only) child node of the `span` element, as that child node represents the header text.

The next task in `clearPreviousResults` is to delete any rows that may already be in the table displaying the search results. Any result rows are child nodes of the `tbody` node, so you start by obtaining a reference to that node using the `document.getElementById` method. Once you have the `tbody` node, you iterate for as long as the `tbody` node has child nodes, where the child nodes are `tr` elements. During each iteration the first child node in the `childNodes` collection is removed from the table body. The iteration ends once no more rows are left in the table body.

The table of search results is built in the `parseResults` function. This function starts by creating a local variable named `results`, which is the XML document retrieved from the XMLHttpRequest object's `responseXML` property.

You use the `getElementsByTagName` method to retrieve all the `property` elements in the XML document as an array and assign the array to the local variable `properties`. Once you have the array of `property` elements, you can iterate over each element in the array and retrieve the property's address, price, and comments.

```
var properties = results.getElementsByTagName("property");
for(var i = 0; i < properties.length; i++) {
    property = properties[i];
    address = property.getElementsByTagName("address")[0].firstChild.nodeValue;
    price = property.getElementsByTagName("price")[0].firstChild.nodeValue;
    comments = property.getElementsByTagName("comments")[0].firstChild.nodeValue;

    addTableRow(address, price, comments);
}
```

Let's examine this iteration closely, as this is the heart of the `parseResults` function. Within the `for` loop the first thing that happens is you get the next element in the array and assign it to the local property named `property`. Then, for each of the child elements in which you're interested—`address`, `price`, and `comments`—you retrieve their respective node values.

Consider the address element, which is a child of the property element. You first get the single address element by calling the getElementsByTagName method on the property element. The getElementsByTagName method returns an array, but since you know that one and only one address element exists, you can reference it using the [0] notation.

Continuing your way down the XML structure, you now have a reference to the address tag, and you now need to get its textual content. Remembering that text is actually a child node of the parent element, you can access the text node of the address element by using the firstChild property. Now that you have the text node, you can retrieve the text by referring to the nodeValue property of the text node.

You use the same process to retrieve the values of the price and comments elements, and each value is assigned to the price and comments local variables, respectively. The address, price, and comments are then passed to a helper function named addTableRow that actually builds a table row with the results data.

The addTableRow function uses W3C DOM methods and JavaScript to build a table row. A row object is created by using the document.createElement method. After creating the row object, a cell object is created for each of the address, price, and comments values using a helper function named createCellWithText. The createCellWithText function creates and returns a cell object with the specified text as the cell's contents.

The createCellWithText function starts by creating a td element by using the document.createElement method. A text node containing the desired text is then created using the document.createTextNode method, and the resulting text node is appended to the td element. The function then returns the newly created td element to the calling function.

The addTableRow function repeats a call to the createCellWithText function for the address, price, and comments value, each time appending the newly created td element to the tr element. Once all the cells have been added to the row, the row is added to the table's tbody element.

That's all! You've successfully read the XML document returned by the server and dynamically created a table of results. Listing 3-6 shows the complete JavaScript and eXtensible HTML (XHTML) code for this example.

Listing 3-6. dynamicContent.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Dynamically Editing Page Content</title>

<script type="text/javascript">
var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}
```

```
function doSearch() {
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET", "dynamicContent.xml", true);
    xmlHttp.send(null);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            clearPreviousResults();
            parseResults();
        }
    }
}

function clearPreviousResults() {
    var header = document.getElementById("header");
    if(header.hasChildNodes()) {
        header.removeChild(header.childNodes[0]);
    }

    var tableBody = document.getElementById("resultsBody");
    while(tableBody.childNodes.length > 0) {
        tableBody.removeChild(tableBody.childNodes[0]);
    }
}

function parseResults() {
    var results = xmlHttp.responseXML;

    var property = null;
    var address = "";
    var price = "";
    var comments = "";

    var properties = results.getElementsByTagName("property");
    for(var i = 0; i < properties.length; i++) {
        property = properties[i];
        address = property.getElementsByTagName("address")[0].firstChild.nodeValue;
        price = property.getElementsByTagName("price")[0].firstChild.nodeValue;
        comments = property.getElementsByTagName("comments")[0]
                                                          .firstChild.nodeValue;

        addTableRow(address, price, comments);
    }
```

```
    var header = document.createElement("h2");
    var headerText = document.createTextNode("Results:");
    header.appendChild(headerText);
    document.getElementById("header").appendChild(header);

    document.getElementById("resultsTable").setAttribute("border", "1");
}


function addTableRow(address, price, comments) {
    var row = document.createElement("tr");
    var cell = createCellWithText(address);
    row.appendChild(cell);

    cell = createCellWithText(price);
    row.appendChild(cell);

    cell = createCellWithText(comments);
    row.appendChild(cell);

    document.getElementById("resultsBody").appendChild(row);
}

function createCellWithText(text) {
    var cell = document.createElement("td");
    var textNode = document.createTextNode(text);
    cell.appendChild(textNode);

    return cell;
}
</script>
</head>

<body>
  <h1>Search Real Estate Listings</h1>

  <form action="#">
    Show listings from
        <select>
            <option value="50000">$50,000</option>
            <option value="100000">$100,000</option>
            <option value="150000">$150,000</option>
        </select>
        to
```

```
        <select>
            <option value="100000">$100,000</option>
            <option value="150000">$150,000</option>
            <option value="200000">$200,000</option>
        </select>
    <input type="button" value="Search" onclick="doSearch();"/>
  </form>



  <span id="header">

  </span>

  <table id="resultsTable" width="75%" border="0">
    <tbody id="resultsBody">
    </tbody>
  </table>
</body>
</html>
```

Sending Request Parameters

So far you've seen how you can use Ajax techniques to send requests to the server and the various ways the client can parse the server's response. The only thing missing in the previous examples is that you're not sending any data as part of the request to the server. For the most part, sending a request to the server without any request parameters doesn't do much good. Without any request parameters, the server has no contextual data by which to create a "personalized" response for the client, and in essence the server will send the identical response to every client.

Unlocking the real power of Ajax techniques requires that you send some contextual data to the server. Imagine an input form that includes a section for entering mailing addresses. You can use Ajax techniques to prepopulate the name of the city that corresponds to the ZIP code entered by the user. Of course, to look up the ZIP code's city, the server needs to know the ZIP code entered by the user.

Somehow you need to pass the ZIP code value entered by the user to the server. Fortunately, the XMLHttpRequest object works much the same as the old HTTP techniques you're used to working with: `GET` and `POST`.

The `GET` method passes the value as name/value pairs as part of the request URL. The end of the resource URL ends with a question mark (?), and after the question mark are the name/value pairs. The name/value pairs are in the form `name=value`, and they are separated by an ampersand (&).

The following line is an example of a `GET` request. The request is sending two parameters, `firstName` and `middleName`, to the application named `yourApp` on the server named `localhost`.

Note that the resource URL and the parameters set are separated by a question mark, and `firstName` and `middleName` are separated by an ampersand:

```
http://localhost/yourApp?firstName=Adam&middleName=Christopher
```

The server knows how to retrieve the named parameters in the URL. Most modern server-side programming environments provide simple APIs, allowing easy access to the named parameters.

The `POST` method of sending named parameters to the server is nearly identical to the `GET` method. The `POST` method, like the `GET` method, encodes the parameters as name/value pairs in the form `name=value`, with each name/value pair separated by an ampersand. The main difference between the two is that the `POST` method sends the parameter string within the request body rather than appending it to the URL the way the `GET` method does.

The HTML usage specification technically recommends that you use the `GET` method when the data processing does not change a data model's state, effectively meaning that you should use the `GET` method when retrieving data. The `POST` method is recommended for operations that change a data model's state, perhaps by storing or updating data or by sending an e-mail.

Each method has its subtle advantages. Since the parameters of a `GET` request are encoded in the request URL, you can bookmark the URL in a browser and easily repeat the request later. In the context of asynchronous requests, though, this doesn't provide much usefulness. The `POST` request is more flexible in terms of the amount of data that can be sent to the server. The amount of data that can be sent using a `GET` request is typically a fixed amount that varies among different browsers, while the `POST` method can send any amount of data.

The HTML `form` element allows you to dictate the desired method by setting the element's method attribute to `GET` or `POST`. The `form` element automatically encodes the data of its input elements according to its method attribute's rules when the form is submitted. The XMLHttpRequest object does not have such built-in behavior. Instead, the developer is responsible for creating the query string containing the data to be sent to the server as part of the request using JavaScript. The techniques for creating the query string are identical regardless of whether you use a `GET` or `POST` request. The only difference is that when sending the request using `GET`, the query string will be appended to the request URL, while with `POST` the query string is sent when calling the XMLHttpRequest object's `send()` method.

Figure 3-4 shows a sample page that demonstrates how to send request parameters to the server. It's a simple input form asking for a first name, middle name, and birthday. The form has two buttons. Each button sends the first name, middle name, and birthday data to the server, one using the `GET` method and one using the `POST` method. The server responds by echoing the input data. The cycle completes when the browser prints the server's response on the page.

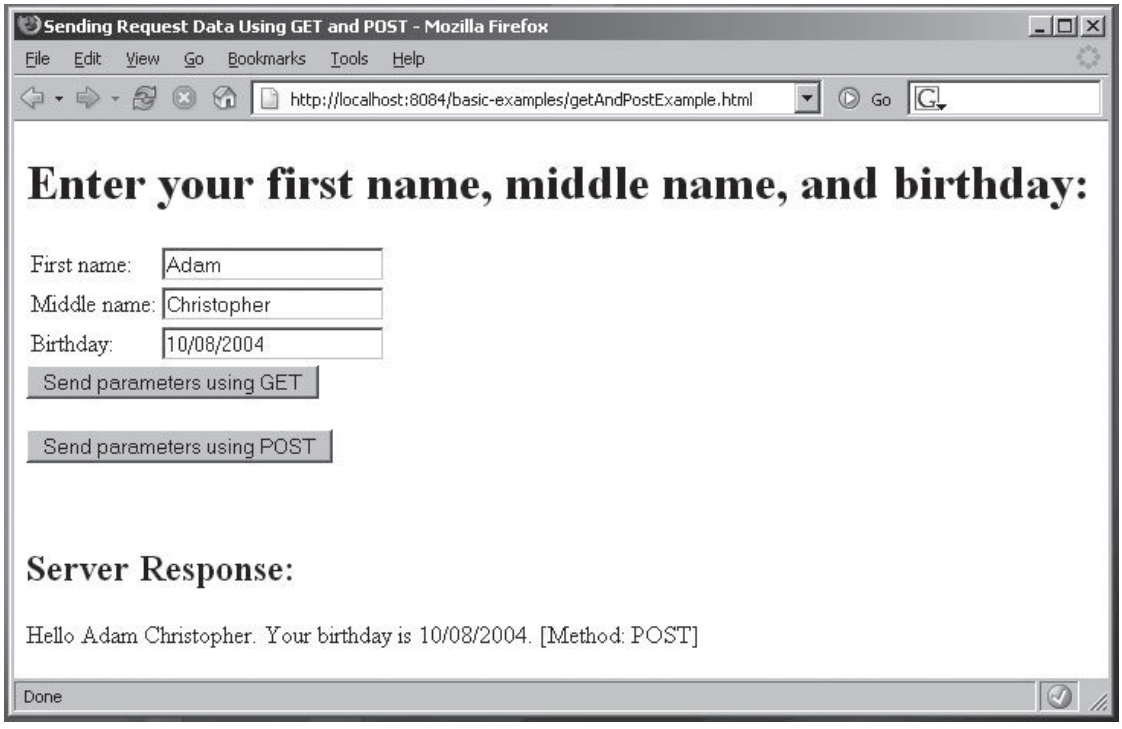

Figure 3-4. *The browser sends the input data using either a* `GET` *or a* `POST` *method, and the server responds by echoing the input data.*

Listing 3-7 shows `getAndPostExample.html`, and Listing 3-8 shows the Java servlet that echoes the first name, middle name, and birthday back to the browser.

Listing 3-7. `getAndPostExample.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Sending Request Data Using GET and POST</title>

<script type="text/javascript">
var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}
```

```
function createQueryString() {
    var firstName = document.getElementById("firstName").value;
    var middleName = document.getElementById("middleName").value;
    var birthday = document.getElementById("birthday").value;

    var queryString = "firstName=" + firstName + "&middleName=" + middleName
        + "&birthday=" + birthday;

    return queryString;
}

function doRequestUsingGET() {
    createXMLHttpRequest();

    var queryString = "GetAndPostExample?";
    queryString = queryString + createQueryString()
        + "&timeStamp=" + new Date().getTime();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET", queryString, true);
    xmlHttp.send(null);
}

function doRequestUsingPOST() {
    createXMLHttpRequest();

    var url = "GetAndPostExample?timeStamp=" + new Date().getTime();
    var queryString = createQueryString();

    xmlHttp.open("POST", url, true);
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.setRequestHeader("Content-Type",
                  "application/x-www-form-urlencoded;");
    xmlHttp.send(queryString);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            parseResults();
        }
    }
}

function parseResults() {
    var responseDiv = document.getElementById("serverResponse");
    if(responseDiv.hasChildNodes()) {
        responseDiv.removeChild(responseDiv.childNodes[0]);
    }
```

```
    var responseText = document.createTextNode(xmlHttp.responseText);
    responseDiv.appendChild(responseText);
}

</script>
</head>

<body>
  <h1>Enter your first name, middle name, and birthday:</h1>

  <table>
    <tbody>
        <tr>
            <td>First name:</td>
            <td><input type="text" id="firstName"/>
        </tr>
        <tr>
            <td>Middle name:</td>
            <td><input type="text" id="middleName"/>
        </tr>
        <tr>
            <td>Birthday:</td>
            <td><input type="text" id="birthday"/>
        </tr>
    </tbody>

  </table>

  <form action="#">
    <input type="button" value="Send parameters using GET"
             onclick="doRequestUsingGET();"/>

    <br/><br/>
    <input type="button" value="Send parameters using POST"
            onclick="doRequestUsingPOST();"/>
  </form>

  <br/>
  <h2>Server Response:</h2>

  <div id="serverResponse"></div>

</body>
</html>
```

Listing 3-8. *Echoing the First Name, Middle Name, and Birthday Back to the Browser*

```
package ajaxbook.chap3;

import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetAndPostExample extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response, String method)
    throws ServletException, IOException {

        //Set content type of the response to text/xml
        response.setContentType("text/xml");

        //Get the user's input
        String firstName = request.getParameter("firstName");
        String middleName = request.getParameter("middleName");
        String birthday = request.getParameter("birthday");

        //Create the response text
        String responseText = "Hello " + firstName + " " + middleName
                + ". Your birthday is " + birthday + "."
                + " [Method: " + method + "]";

        //Write the response back to the browser
        PrintWriter out = response.getWriter();
        out.println(responseText);

        //Close the writer
        out.close();
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        //Process the request in method processRequest
        processRequest(request, response, "GET");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        //Process the request in method processRequest
        processRequest(request, response, "POST");
    }
}
```

Let's examine the server-side code first. This example uses a Java servlet to handle the request, although you can use any server-side technology such as PHP, CGI, or .NET. Java servlets must define a `doGet` method and a `doPost` method, with each method being called according to the request method. In this example, both `doGet` and `doPost` will call the same method, `processRequest`, to handle the request.

The `processRequest` method starts by setting the content type of the response to text/xml, even though in this example XML isn't actually used. The three input fields are retrieved from the `request` object by using the `getParameter` method. A simple sentence is built using the first name, middle name, and birthday, along with the type of request method. The sentence is then written to the response output stream, and finally the response output stream is closed.

The browser-side JavaScript is again similar to previous examples but with a few added twists. A utility function named `createQueryString` is responsible for encoding the input parameters as a query string. The `createQueryString` function simply retrieves the input values for the first name, middle name, and birthday and appends them as name/value pairs. Each name/value pair is separated by an ampersand. This function returns the query string, allowing it to be reused by both the `GET` and `POST` operations.

Clicking the Send Parameters Using GET button calls the `doRequestUsingGET` function. This function, like many of the previous examples, starts by calling the function that creates an instance of the XMLHttpRequest object. Next, the query string that encodes the input values is created.

The request endpoint for this example is the servlet named `GetAndPostExample`. The query string is created by concatenating the query string returned by the `createQueryString` function to the request endpoint, separated by a question mark.

The JavaScript continues as you've seen before. The XMLHttpRequest object's `onreadystatechange` property is set to use the `handleStateChange` function. The `open()` method specifies that this is a `GET` request and the endpoint URL, which in this case contains the encoded parameters. The `send()` method sends the request to the server, and the `handleStateChange` function handles the server response.

The `handleStateChange` function calls the `parseResults` function upon successful completion of the request. The `parseResults` function retrieves the `div` element that contains the server's response and stores it in a local variable named `responseDiv`. Any previous server results are first removed from `responseDiv` by using its `removeChild` method. Finally, a new text node is created containing the server's response and is appended to `responseDiv`.

The techniques for using the `POST` method instead of `GET` are identical except for how the request parameters are sent to the server. Recall that when using `GET`, the name/value pairs are appended to the destination URL. The `POST` method sends the same query string as part of the request body.

Clicking the Send Parameters Using POST button calls the `doRequestUsingPOST` function. Like the `doRequestUsingGET` function, it starts by creating an instance of the XMLHttpRequest object. The script continues by creating the query string that contains the parameters to be sent to the server. Note that the query string is not concatenated with the destination URL.

The XMLHttpRequest object's `open()` method is invoked, this time specifying `POST` as the request method in addition to specifying the "bare" destination URL. The `onreadystatechange` property is set to the `handleStateChange` function, allowing the response to be processed in the same way as the `GET` method. To ensure that the server knows that the request parameters can be found within the request body, the `setRequestHeader` is called, setting the `Content-Type`

value to `application/x-www-form-urlencoded`. Lastly, the `send()` method is invoked by passing the query string as the method parameter.

The output of clicking the two buttons is identical. A string echoing the specified first name, middle name, and birthday is displayed on the page along with the type of request method that was used.

WHY IS THE TIMESTAMP APPENDED TO THE DESTINATION URL?

Under some conditions, some browsers cache the results of multiple XMLHttpRequest requests to the same URL. This can cause undesirable results if the response could be different for each request. Appending the current timestamp to the end of the URL ensures the uniqueness of the URL and thus prevents browsers from caching the results.

Sending Request Parameters As XML

Compatibility of JavaScript amongst today's modern browsers is light years ahead of where it was even a few years ago. Combined with the increasing sophistication of JavaScript development tools and techniques, you may decide to begin using the Web browser as a development platform. Instead of merely relying on the browser to serve as the View in the Model-View-Controller pattern, you may decide to implement part of the business model in JavaScript. You can use Ajax techniques to persist changes in the model to the backend server. If the model is kept on the browser, then changes to the model can be communicated to the server en masse, reducing the number of remote calls made to the server and possibly increasing performance.

A simple query string consisting of name/value pairs is probably not robust enough to communicate a large number of possibly complex model changes to the server. A better solution might be to send changes to the model as XML to the server. How can you send XML to the server?

You can send XML to the server as part of the request body, much like the query string is sent as part of the request body during a `POST` request. The server can read XML from the request body and work with it from there.

The following example demonstrates how you can send XML to the server during an Ajax request. Figure 3-5 shows the page, which is a simple select box in which the user selects the types of pets the user has. It's a simplistic example, but it shows how you can send XML to the server.

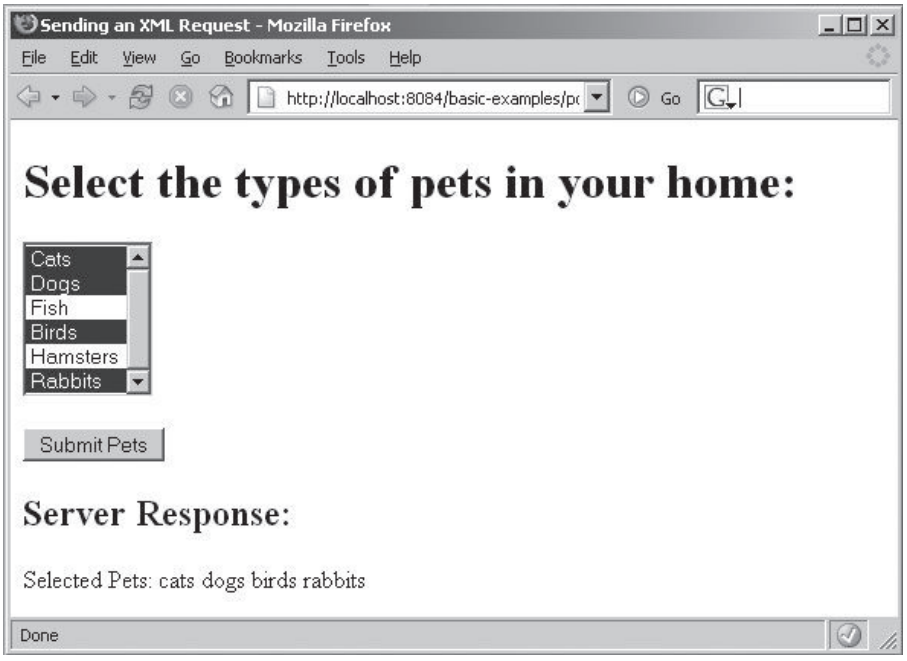

Figure 3-5. *The items selected in the select box are sent to the server as XML.*

Listing 3-9 shows `postingXML.html`.

Listing 3-9. `postingXML.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Sending an XML Request</title>

<script type="text/javascript">

var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}

function createXML() {
    var xml = "<pets>";
```

```
    var options = document.getElementById("petTypes").childNodes;
    var option = null;
    for(var i = 0; i < options.length; i++) {
        option = options[i];
        if(option.selected) {
            xml = xml + "<type>" + option.value + "<\/type>";
        }
    }

    xml = xml + "<\/pets>";
    return xml;
}

function sendPetTypes() {
    createXMLHttpRequest();

    var xml = createXML();
    var url = "PostingXMLExample?timeStamp=" + new Date().getTime();

    xmlHttp.open("POST", url, true);
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded;");
    xmlHttp.send(xml);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            parseResults();
        }
    }
}

function parseResults() {
    var responseDiv = document.getElementById("serverResponse");
    if(responseDiv.hasChildNodes()) {
        responseDiv.removeChild(responseDiv.childNodes[0]);
    }

    var responseText = document.createTextNode(xmlHttp.responseText);
    responseDiv.appendChild(responseText);
}


</script>
</head>
```

```
<body>
  <h1>Select the types of pets in your home:</h1>

  <form action="#">
    <select id="petTypes" size="6" multiple="true">
        <option value="cats">Cats</option>
        <option value="dogs">Dogs</option>
        <option value="fish">Fish</option>
        <option value="birds">Birds</option>
        <option value="hamsters">Hamsters</option>
        <option value="rabbits">Rabbits</option>
    </select>

    <br/><br/>
    <input type="button" value="Submit Pets" onclick="sendPetTypes();"/>
  </form>

  <h2>Server Response:</h2>

  <div id="serverResponse"></div>

</body>
</html>
```

This example works much the same as the earlier `POST` example. The difference is that instead of a query string composed of name/value pairs being sent, an XML string is sent to the server.

Clicking the Submit Pets button on the form invokes the `sendPetTypes` function. Like previous examples, this function first creates an instance of the XMLHttpRequest object. It then calls a helper function named `createXML` that builds the XML string of the selected pet types.

The function `createXML` retrieves a reference to the `select` element using the `document.getElementbyId` method. It then iterates over all its child `option` elements and for each selected option creates the XML tag for the selected pet type and appends it to the rest of the XML. At the end of the iteration, the closing `pets` tag is appended to the XML string before it's returned to the calling function.

Once it has obtained the XML string, the `sendPetTypes` function continues by preparing XMLHttpObject for the request and then sends the XML to the server by specifying the XML string as the parameter to the `send()` method.

IN THE CREATEXML METHOD, WHY DOES A BACKSLASH PRECEDE THE FORWARD SLASH IN THE CLOSE TAGS?

Thanks to a quirk in the SGML specification from which HTML is derived, end tags are recognized within `script` elements, but other content such as start tags and comments are not. The backslash prevents the string from being parsed from markup. Most browsers will safely handle the instances where a backslash is omitted, but to stick to the strict XHTML standard, use a backslash.

The astute reader will notice that according to the documentation for the XMLHttpRequest object, the `send()` method can take both a string and an instance of an XML document object. Why, then, does this example use string concatenation to create the XML instead of creating document and element objects? Unfortunately, at this time no cross-browser technique exists for building a document object from scratch. Internet Explorer exposes this functionality through ActiveX objects, Mozilla browsers expose it through native JavaScript objects, and other browsers either don't support it at all or support it via other means.

The server-side code to read the XML, shown in Listing 3-10, is a bit more complicated. This example uses a Java servlet to read the request and parse the XML string, although you could also use other server-side technologies.

The servlet's `doPost` method is invoked upon receipt of the XMLHttpRequest object's request. The `doPost` method uses a helper method named `readXMLFromRequestBody` to extract the XML from the body of the request. The `doPost` method then converts the XML string into a `Document` object using the JAXP interfaces.

Note that the `Document` object is an instance of the `Document` interface specified by the W3C. As such, it has the same methods as the browser's `document` object such as `getElementsByTagName`. You use this method to get a list of all the `type` elements in the document. For each `type` element in the document, the text value is obtained (remember that the text value is the first child node of the `type` element) and appended to a string. After all the `type` elements have been consumed, the response string is written back to the browser.

Listing 3-10. `PostingXMLExample.java`

```
package ajaxbook.chap3;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class PostingXMLExample extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

        String xml = readXMLFromRequestBody(request);
        Document xmlDoc = null;
        try {
            xmlDoc =
                    DocumentBuilderFactory.newInstance().newDocumentBuilder()
                    .parse(new ByteArrayInputStream(xml.getBytes()));
        }
```

```
        catch(ParserConfigurationException e) {
            System.out.println("ParserConfigurationException: " + e);
        }
        catch(SAXException e) {
            System.out.println("SAXException: " + e);
        }

        /* Note how the Java implementation of the W3C DOM has the same methods
         * as the JavaScript implementation, such as getElementsByTagName and
         * getNodeValue.
         */
        NodeList selectedPetTypes = xmlDoc.getElementsByTagName("type");
        String type = null;
        String responseText = "Selected Pets: ";
        for(int i = 0; i < selectedPetTypes.getLength(); i++) {
           type = selectedPetTypes.item(i).getFirstChild().getNodeValue();
           responseText = responseText + " " + type;
        }

        response.setContentType("text/xml");
        response.getWriter().print(responseText);
    }

    private String readXMLFromRequestBody(HttpServletRequest request){
        StringBuffer xml = new StringBuffer();
        String line = null;
        try {
            BufferedReader reader = request.getReader();
            while((line = reader.readLine()) != null) {
                xml.append(line);
            }
        }
        catch(Exception e) {
            System.out.println("Error reading XML: " + e.toString());
        }
        return xml.toString();
    }
}
```

Sending Data to the Server Using JSON

Now that you've been working with JavaScript a little more, you may be thinking to yourself that you want to keep more model information on the browser. However, after seeing the previous example that uses XML to send complex data structures to the server, you may be less inclined to do so. Creating XML strings by string concatenation is not an appealing or robust technique for generating or modifying XML data structures.

Overview of JSON

An alternative to XML is JSON, which you can find at `www.json.org`. JSON is a text format that is language independent but that uses conventions similar to the C family of languages such as C, C#, JavaScript, and others. JSON is built on the following two data structures that are supported by virtually all modern programming languages:

- A collection of name/value pairs. In modern programming languages these are realized as an object, record, or dictionary.
- An ordered list of values, which is usually realized as an array.

Since these structures are supported by so many programming languages, JSON is an ideal choice as a data interchange format between disparate systems. Additionally, since JSON is based on a subset of standard JavaScript, it should be compatible in all modern Web browsers.

A JSON object is an unordered set of name/value pairs. The object begins with a { and ends with a }. A colon separates the name/value pairs. A JSON array is an ordered collection of values that begins with a [and ends with a]. A comma separates the values of the array. A value can be a string (enclosed in double quotes), a number, a `true` or `false`, an object, or an array. This allows structures to be nested. Figure 3-6 is a good visual guide for describing the makeup of a JSON object.

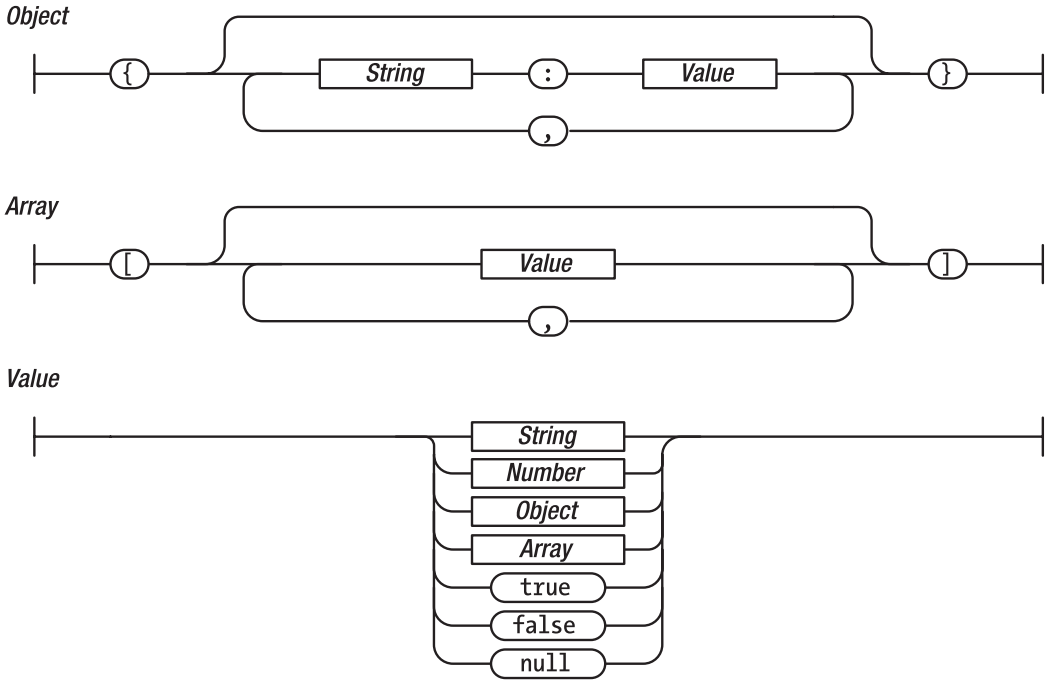

Figure 3-6. *Graphical representation of a JSON object structure (courtesy of* `www.json.org`*)*

Consider the simple example of an `Employee` object. An `Employee` object might consist of data such as a first name, last name, employee number, and title. Using JSON, you represent an instance of the `Employee` object like this:

```
var employee = {
    "firstName" : John
    , "lastName" : Doe
    , "employeeNumber" : 123
    , "title" : "Accountant"
}
```

You can then use the object's properties using the standard dot notation, like so:

```
var lastName = employee.lastName;  //Access the last name
var title = employee.title;        //Access the title
employee.employeeNumber = 456;     //Change the employee number
```

JSON prides itself on being a lightweight data interchange format. The same `Employee` object described as XML might look like this:

```
<employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
    <employeeNumber>123</employeeNumber>
    <title>Accountant</title>
</employee>
```

Clearly, the JSON encoding is smaller than the XML encoding. The smaller size of the JSON encoding could potentially make a big performance difference when sending large amounts of data over a network.

The Web site at `www.json.org` lists at least 14 bindings to other programming languages, meaning no matter what technology you use on the server, you should be able to communicate with the browser through JSON.

Example Using JSON

The following example is a simple demonstration of how to use JSON to translate JavaScript objects to a string format and send the string to the server using Ajax techniques; the server will then create an object based on the string. The example has no business logic and little user interaction, as it focuses on the JSON techniques both on the client side and on the server side. Figure 3-7 shows a "stringified" `Car` object.

Because this example is nearly identical to the previous `POST` examples, we'll focus on the JSON-specific techniques. Clicking the form button invokes the `doJSON` function. This function first asks the `getCarObject` function to return a new instance of a `Car` object. You then use the JSON JavaScript library (freely available from `www.json.org`) to translate the `Car` object into a JSON string that is then displayed in an alert box. The JSON-encoded `Car` object is then sent to the server using the XMLHttpRequest object.

Thanks to the freely available JSON-to-Java library, coding the Java servlet that services the JSON request is simple. Better yet, since JSON bindings are available for every flavor of server technology, you can easily implement this example using any server-side technology.

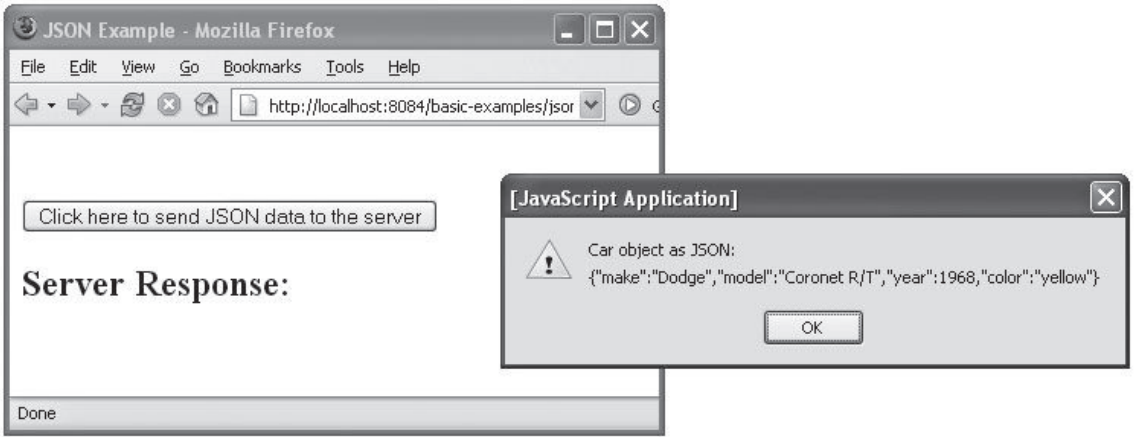

Figure 3-7. *The "stringified"* `Car` *object*

The `JSONExample` servlet's `doPost` method services the JSON request. It first asks the `readJSONStringFromRequestBody` method to obtain the JSON string from the request body. It then creates an instance of `JSONObject`, supplying the JSON string to the object constructor. The `JSONObject` automatically parses the JSON string at object creation time. Once `JSONObject` is created, you can use the various `get` methods to retrieve the object properties in which you're interested.

You use the `getString` and `getInt` methods to retrieve the year, make, model, and color properties. These properties are concatenated to form the string that is returned to the browser and displayed on the page. Figure 3-8 shows the response from the server after reading the JSON object.

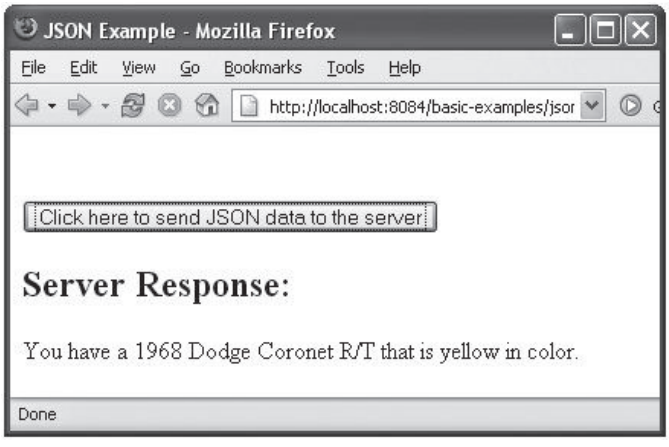

Figure 3-8. *The response from the server after reading the JSON string*

Listing 3-11 shows `jsonExample.html`, and Listing 3-12 shows `JSONExample.java`.

Listing 3-11. jsonExample.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JSON Example</title>

<script type="text/javascript" src="json.js"></script>
<script type="text/javascript">

var xmlHttp;

function createXMLHttpRequest() {
    if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
    }
}

function doJSON() {
    var car = getCarObject();

    //Use the JSON JavaScript library to stringify the Car object
    var carAsJSON = JSON.stringify(car);
    alert("Car object as JSON:\n " + carAsJSON);

    var url = "JSONExample?timeStamp=" + new Date().getTime();

    createXMLHttpRequest();
    xmlHttp.open("POST", url, true);
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.setRequestHeader("Content-Type",
                    "application/x-www-form-urlencoded;");
    xmlHttp.send(carAsJSON);
}

function handleStateChange() {
    if(xmlHttp.readyState == 4) {
        if(xmlHttp.status == 200) {
            parseResults();
        }
    }
}
```

```
function parseResults() {
    var responseDiv = document.getElementById("serverResponse");
    if(responseDiv.hasChildNodes()) {
        responseDiv.removeChild(responseDiv.childNodes[0]);
    }

    var responseText = document.createTextNode(xmlHttp.responseText);
    responseDiv.appendChild(responseText);
}

function getCarObject() {
    return new Car("Dodge", "Coronet R/T", 1968, "yellow");
}

function Car(make, model, year, color) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
}

</script>
</head>

<body>

  <br/><br/>
  <form action="#">
      <input type="button" value="Click here to send JSON data to the server"
        onclick="doJSON();"/>
  </form>

  <h2>Server Response:</h2>

  <div id="serverResponse"></div>

</body>
</html>
```

Listing 3-12. JSONExample.java

```
package ajaxbook.chap3;

import java.io.*;
import java.net.*;
import java.text.ParseException;
import javax.servlet.*;
import javax.servlet.http.*;
import org.json.JSONObject;
```

```
public class JSONExample extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        String json = readJSONStringFromRequestBody(request);

        //Use the JSON-Java binding library to create a JSON object in Java
        JSONObject jsonObject = null;
        try {
            jsonObject = new JSONObject(json);
        }
        catch(ParseException pe) {
            System.out.println("ParseException: " + pe.toString());
        }

        String responseText = "You have a " + jsonObject.getInt("year") + " "
            + jsonObject.getString("make") + " " + jsonObject.getString("model")
            + " " + " that is " + jsonObject.getString("color") + " in color.";

        response.setContentType("text/xml");
        response.getWriter().print(responseText);
    }

    private String readJSONStringFromRequestBody(HttpServletRequest request){
        StringBuffer json = new StringBuffer();
        String line = null;
        try {
            BufferedReader reader = request.getReader();
            while((line = reader.readLine()) != null) {
                json.append(line);
            }
        }
        catch(Exception e) {
            System.out.println("Error reading JSON string: " + e.toString());
        }
        return json.toString();
    }
}
```

Summary

In this chapter, you explored the myriad methods by which the XMLHttpRequest object and server can communicate with each other. The XMLHttpRequest object can send requests using either the HTTP `GET` or `POST` methods, while the request data can be sent as a query string, XML, or JSON data. After handling the request, the server will typically respond by sending simple text, XML data, or even JSON data. Each format may be best suited for certain situations.

Ajax wouldn't be particularly useful if you had no way to dynamically update the page's contents based on the results of a request. Today's modern browsers expose the contents of a Web page as an object model in accordance with the W3C DOM standards. This object model allows scripting languages such as JavaScript to add, update, and remove content on the page without the need for a round-trip to the server. While some quirks still exist, most Web pages written according to W3C standards and modified using standard JavaScript will behave identically in any standards-compliant browser. Most of today's modern browsers also support the nonstandard `innerHTML` property, which can be used to update the elements on a Web page.

You are now familiar with the XMLHttpRequest object and how you can use it to seamlessly communicate with the server. You also know how to update the Web page's contents dynamically. What's next?

Chapter 4 offers a glimpse into the endless possibilities now afforded by Ajax. It's one thing to know how to use Ajax, but it's a completely different thing to apply it in a useful setting. The next chapter introduces situations common to Web applications that are prime candidates for Ajax techniques.